Abstract

BUKSAS, MICHAEL WILLIAM. Modeling, Analysis and Implementation of Forward and Inverse Problems in One Dimensional Electromagnetic Scattering with Differential and Hysteretic Polarization Models (Under the direction of H. Thomas Banks)

We begin with the derivation from Maxwell's Equations of equations describing the interaction of planar electromagnetic waves with dielectric materials. This derivation is completed by the addition of constitutive laws which govern the polarization of the material. We consider two types of constitutive laws expressed as differential equations or as the convolution of the history of the electric field with a hysteresis kernel. These laws are expressed in terms of parameters whose values describe various materials. Our interest here is in contrasting these two formulations for the purposes of simulation and recovery of the values of the parameters. The Debye and Lorentz models are the differential constitutive models of first and second order and are the ones we consider here.

We develop numerical schemes for each formulation of the problem which use Galerkin finite elements to discretize the space variable, creating a differential system of equations. For the hysteresis formulation, another Galerkin approximation is used for the history of the electric field and the approximation of the hysteresis integral. Using these schemes, we simulate brief pulses of high-frequency electromagnetic waves impinging upon a homogeneous slab of material. These simulations demonstrate the same precursor formation found in frequency domain analyses of these problems. We also show that the numerical approximation of the hysteresis formulation accurately reproduces the results of the differential Debye model when the correct hysteresis function is used. This same method is shown to be impractical for the highly oscillatory kernel function for the Lorentz model.

The simulations are used in the inverse problem, where we attempt to recover the values of the parameters from measurements of the electric field reflected off the

surface of the slab. We estimate the values of the model parameters by minimizing the difference between the results of the simulation and a set of data. The data is generated by adding noise of relative magnitude to a result from the same simulation. By comparing the results of the optimization to the parameter values used to generate the data, we measure the effect of the noise. Our results indicate that with certain exceptions, the parameter values can be recovered in the presence of noise of magnitude up to 5% of the magnitude of the signal. The exceptions arise from the sensitivity of the simulation to the various parameters but are compatible with an accurate representation of the dynamics of the polarization. The sensitivity of the result of the Debye model simulation on various parameter values is also shown to depend on the frequency of the interrogating signal. Furthermore, for experiments using the Debye model, a hybrid technique is presented for estimating both the physical parameters and the thickness of the slab. This is done by expanding the collection of data to include part of the interrogating pulse which travels completely through the material and matching the return time of this pulse with the return time observed in the data.

The software written to perform the numerical computations described above is made part of a project to develop a re-usable library of code for scientific computing. We chose the object-oriented language C++ for this purpose with the intention of using specific features to facilitate the development of the library. We document here the features of interest and the resulting library of code. We also discuss some design patterns, supported by the object oriented paradigm which were used in the design of the code.

MODELING, ANALYSIS AND IMPLEMENTAION OF FORWARD AND INVERSE PROBLEMS IN ONE DIMENSIONAL ELECTROMAGNETIC SCATTERING WITH DIFFERENTIAL AND HYSTERETIC POLARIZATION MODELS

BY

Michael William Buksas

A DISSERTION SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARMENT OF MATHEMATICS

RALEIGH

September 1998

APPROVED BY:

CHAIR OF ADVISORY COMMITTEE

Biography

The author was born in Waukegan, Illinois on April 29, 1971 and grew up in Kentucky and Arizona. In 1993 he received a bachelor of science degree in mathematics from Harvey Mudd College, graduating with High Distinction and Honors in Mathematics. He received his doctorate in philosophy in computational mathematics in December 1998 from North Carolina State University and has accepted a post-doctoral position in the theoretical division of Los Alamos National Laboratory.

Acknowledgements

Numerous people have provided both professional and personal support invaluable to the completion of this research. Foremost, I would like to thank Dr. H.T. Banks for his guidance in this project, his words of personal encouragement, and his unending support of me in this work and my career. His example is one that I will never forget.

I also greatly appreciate the efforts of the other members of my committee, Dr. Michael Shearer, Dr. Hien Tran and Dr. Kazifumi Ito for their interest in my work and excellent suggestions for improving this document.

I would also like to thank Dr. Richard Albanese of the Mathematical Products Division, Air Force Research Laboratory for providing the genesis of a fascinating problem. His continued interest and valuable insights throughout this project have contributed greatly to it's success.

I am also grateful to my former associates in the Center for Research in Scientific Computation, Dr. Yun Wang and Dr. Irene Groselj, for participating in an exchange of ideas which developed this project further than it would have otherwise gone. Also, Dr. Mac Hyman of Los Alamos National Laboratory for additional research experiences and opportunities which have opened the door to my future career.

Other people at NCSU have kept me on track these past years. Janet Early and Rory Schell have both saved me from missing important deadlines and I am indebted to them both.

I have recieved financial support during the course of this research from the Air Force Office of Scientific Research (AFOSR) through grants F49620-91-1-0236,

F49620-98-1-0180 and F29620-95-1-0375 and the U.S. Department of Education Graduate Assistance in Areas of National Need (GAANN) Fellowship Program through Grant P220A40730 and the Los Alamos National Laboratory Graduate Research Program.

For personal support I have turned often to my family and friends, who have not once asked me to stop complaining. My heart-felt thanks go to my parents for their boundless support and belief in me. To my fellow students Eliza Berry, Cindy Musante, Melissa and Toni Choi, Rebecca Segal, Ricardo del Rosario, John Peach, Mike Jeffris, Julie Raye, and many others, thank you, these years would have been much less enjoyable without you. I also thank my good friend Scott Zoldi for helping me explore both Northern New Mexico and the wilderness of post-doctoral life, and Brenda Smith for personal encouragement when I needed it most.

My greatest inspiration and encouragement has come from my dear friend, Mary Jenkins, without whom the last few years would have been much less meaningful. For her, I have the deepest gratitude and hope for the future.

Table of Contents

List of Figures

List of Tables

Chapter 1

Background

A survey of the mathematical literature reveals considerable interest in the identification of material parameters describing electromagnetic phenomenon. For our purposes, we categorize the materials and the models employed to describe them as either dispersive or non-dispersive, where dispersive materials are those in which electromagnetic waves of different frequencies have different phase velocities. When modeled in the frequency domain, this is manifested as parameters which depend explicitly on frequency. In time domain models, the same phenomenon can be captured with constitutive laws in which the electric and/or the magnetic polarizations are expressed in terms of the convolution of the history of the electric and magnetic fields. The equivalence of the two in the case of electric polarization dispersion is shown by Jackson [Jac, pp.306]

Forward problems in this field are dominated by one dimensional scattering problems where planar electromagnetic waves impinge on dielectric slabs. In a series of four papers [KK, KK2, KK3, KK4] Kristenson and Krueger examine this problem with a wave splitting technique and the derivation of scattering operators which satisfy imbedding equations. The reconstruction of the functions representing the physical parameters proceeds from the imbedding equations and a scheme is presented which is shown to be robust in the presence of noise. The physical model covers stratified media, meaning that the material is inhomogeneous in the direction of the propagation

of the waves.

A similar approach is applied to a slightly different model by Weston [Wes] who considers a dissipative wave equation equivalent to the problem of planar waves in stratified media. These results are extended by Krueger [Kru, Kru2, Kru3] to cover media in multiple slabs, thus containing multiple discontunities in the material parameters. Corones and Sun [CS] use the same method of wave splitting and invariant imbedding to reconstruct coefficients in a one dimensional wave equation with a source term. In another paper, He and Ström [HS] also consider the scattering problem for stratified materials illuminated with waves generated by a magnetic dipole.

Some progress has also been made into more general geometries, largely due to the increasing sophistication of wave splitting techniques in higher dimensions. In a paper by Weston [Wes2] a decomposition of solutions of the dissipative wave equation in $\mathbb{R}^3$ is given and integro-differential equations are derived for the reflection operator. The reconstruction of the velocity and dissipation coefficients from the kernel of this operator are demonstrated.

Inverse problems involving dissipative materials follow a similar pattern of development in moving from one dimensional scattering with planar waves to more general settings. Beezley and Krueger [BK] began investigation into these problems by employing a method similar to the non-dispersive case in one dimension. Imbedding equations are derived for the reflection operator relating the incident and scattered parts of the split wave solution. The dielectric response kernel is reconstructed from the imbedding equations derived for homogeneous semi-infinite and finite slabs. The reconstruction is also carried out numerically in the presence of noise applied to the reflection kernel.

Well-posedness results for these problems are demonstrated by Bui [Bui] in which a time domain model involving both conductivity and electric polarization are considered. Theorems governing the existence, uniqueness and continuous dependence of solutions for both the forward and inverse problems were given.

In a different setting Sun [Sun] identifies the source current embedded inside a

dispersive material using a time-domain approach. In a paper by Lerche [Ler] a different integral equation is derived relating the dielectric response function to an operator representing the frequency domain absorption characteristics of a material. Wolfersdorf [Wol] extends these results to finite and semi-infinite slabs and derives exact solutions to the integral equations for the dielectric response.

A different geometry is considered by Kreider [Kri] who poses the problem of electromagnetic scattering in a stratified cylinder. The problem is also made one dimensional through angular symmetry of the cylinder and the fields. Unlike other problems with dispersive media, the material is inhomogeneous in space, although the dielectric parameter is restricted to functions which are separable between the variables of space and frequency: $\epsilon(x, \omega) = \epsilon_1(x)\epsilon_2(\omega)$.

Another problem which allows for spatially inhomogeneous dispersive materials is considered in [HFL]. Here an optimization approach is applied to match simulations to real and synthetic data. The constitutive equations permit convolution terms in both the electric polarization and the conductivity. The limitations of simultaneous reconstruction of the spatial and time-varying parts of the kernel functions is considered when both transmission and reflection data are available.

Another inhomogeneous two dimensional problem is considered by Colton and Monk [CM] who use a frequency domain approach in modeling the interaction of electromagnetic waves with human tissue for the detection of leukemia in bone marrow. The geometry of the medium is presumed known and the two dimensional domain is further partitioned into sub-domains of known geometry. The only unknowns are the constitutive parameters describing the bone marrow, and these are considered functions of space as well. The reconstruction of the bone marrow parameters is demonstrated and is shown to be robust in the presence of relative noise with magnitude as high as 1%.

Another problem motivated by an application is given in [RGKM] where models for the penetration of radar waves in soil are given. The ground is represented by a stratified semi-infinite slab and electromagnetic signals are generated by a circular

loop of current. Well-posedness results are given for the inverse problem.

In summary, we note the differences between the physical models and the solution methods employed in these papers and our results which follow. A number of these papers differ from ours in that they deal with non-dispersive materials. This eliminates the need to reconstruct parameters which are functions of frequency or history kernels which are functions of time. These papers do consider materials which are inhomogeneous in the space variable, however, and while the model formulation is in the time domain, the forward and inverse problems are solved by wave-splitting and invariant imbedding techniques. A number of these papers consider physical problems similar to ours, in which planar electromagnetic waves impinge normally on slabs of material but progress has also been made into three dimensional settings for non-dispersive problems.

Among the papers which employ dispersive models, many differ from ours by using a frequency domain approach which is best suited for physical problems involving time-harmonic solutions. Among those treatments which use time-domain models, most use the wave splitting approach popular with the non-dispersive materials to formulate the forward and inverse problems. Only one paper [HFL] used an optimization approach and formulated an inverse problem using the time domain data itself. While this paper and others considered inhomogeneous materials, the geometry of the material slab is considered to be known. This differs from our problem in that we do not assume the thickness of the material slab is known *a priori* and attempt to identify this dimension along with the parameters describing the electromagnetic behavior.

Chapter 2

Introduction and Problem Formulation

2.1 Motivation and the General Inverse Problem

The ability to interrogate the interior of tissues and other materials has applications to medical imaging and the early detection of anomalies. For example, the results we will present here suggest that the use of a metal tip catheter, threaded into a target organ such as the colon or arterial system, will provide the necessary localization of interrogating probes to allow practical diagnostics involving geometry. Furthermore, it is hoped that the *in vivo* dielectric properties of tissues and organs can be correlated with metabolic functioning. Hence the accurate determination of these dielectric properties can be employed in the evaluation of functional integrity of tissues and organs in subjects.

Other suggested applications for the technology developed here are mine, ordinance and camouflage detection, non-destructive damage detection in aircraft, and subsurface and atmospheric environmental monitoring.

The goals of electromagnetic interrogation as presented here are twofold: the determination of both the geometry and dielectric properties of the materials under investigation. We consider the generalized problem depicted schematically in Figure 2.1. The domain Ω of the object under consideration has both a known and unknown portion of the boundary. The unknown portion $\Gamma(q)$ is presumed to be

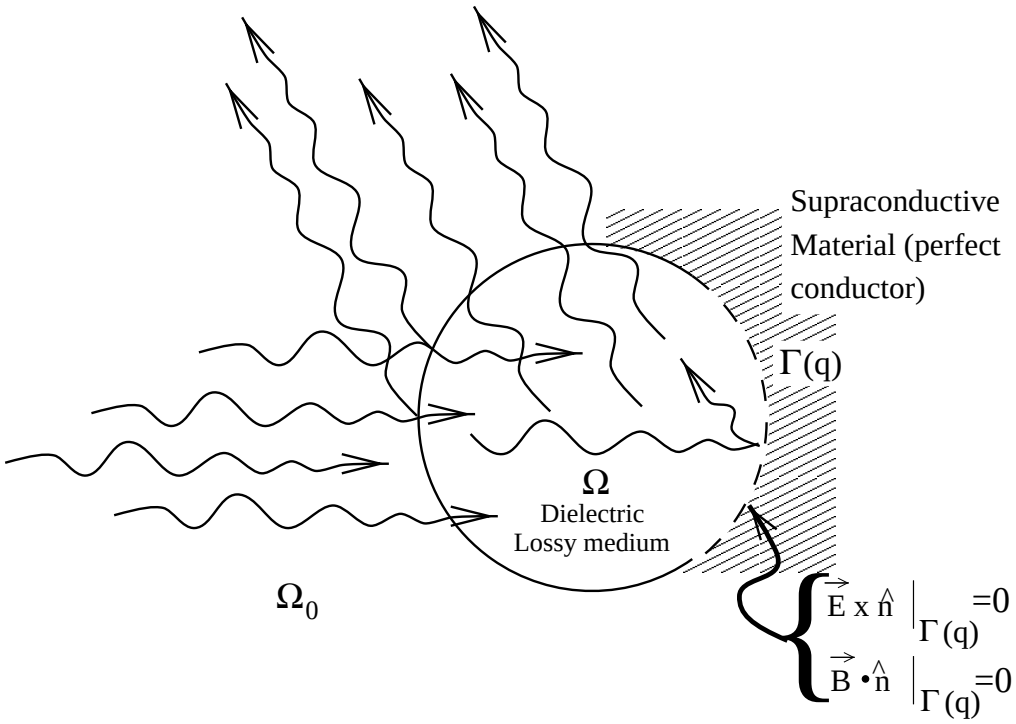

Figure 2.1: Schematic Diagram of Geometry

backed by a supraconductive material with an effectively infinite conductivity. On this boundary with outer normal $\hat{n}$, we thus have $\vec{E} \times \hat{n} = 0$ and $\vec{B} \cdot \hat{n} = 0$ [Bal, p20]. Note that the unknown nature of the boundary is represented by its dependence on a set of parameters q which are to be determined to establish the geometry of the object.

The electric and magnetic fields inside Ω and exterior to Ω (this region will be denoted Ω_0) are governed by the macroscopic Maxwell's equations [Jac, Bal, St]. To describe the electromagnetic behavior of complex materials, we express Maxwell's equations in a general form which includes terms for electric and magnetic polarization. We have

$$\begin{aligned} \nabla\times\vec{E} &= -\frac{\partial\vec{B}}{\partial t} \\ \nabla\times\vec{H} &= \frac{\partial\vec{D}}{\partial t}+\vec{J} \\ \nabla\cdot\vec{D} &= \rho \\ \nabla\cdot\vec{B} &= 0. \end{aligned} \qquad \begin{aligned} \vec{D} &= \epsilon_0\vec{E}+\vec{P} \\ \vec{B} &= \mu_0\vec{H}+\mu_0\vec{M} \\ \vec{J} &= \vec{J}_c+\vec{J}_s \end{aligned}$$

The vector-valued functions $\vec{E}$ and $\vec{H}$ represent the strengths of the electric and magnetic fields respectively, while $\vec{D}$ and $\vec{B}$ are the electric and magnetic flux densities respectively. The two current contributions are denoted by $\vec{J}_c$, the conduction current density, and $\vec{J}_s$, a source current density. The electric and magnetic polarizations are represented by $\vec{P}$ and $\vec{M}$, respectively . The scalar quantity ρ represents the density of electric charges unaccounted for in the electric polarization.

The three quantities $\vec{M}, \vec{P}$ and $\vec{J}_c$ embody the behavior of the material in response to the electromagnetic fields. Additional material-dependent equations (constitutive laws) are required to determine their dependence on the components of the fields $\vec{E}$ and $\vec{H}$. These are also necessary to make the system of equations completely determined. There are twenty-two unknowns in the equations above, three components of the seven vector fields $\vec{E}, \vec{H}, \vec{D}, \vec{B}, \vec{P}, \vec{M}, \vec{J}_c$ and the scalar quantity ρ. The equations above provide fourteen total equations and the three constitutive laws for the quantities $\vec{M}, \vec{P}$ and $\vec{J}_c$ add nine more. The dependence of these equations on the material is reflected both in the choice of a mathematical model and in the parameters (possibly functions) appearing in the model. Estimation of these parameters are the goals in the inverse problems we formulate below.

The region Ω_0 external to the medium is treated as empty space and is devoid of conductivity or polarization effects, hence $\vec{M} = 0, \vec{P} = 0$ and $\vec{J}_c = 0$ in Ω_0, and all of the necessary parameters for the determination of the fields are assumed known in this domain. The source current density term $\vec{J}_s$ will also be non-zero only at points in Ω_0. The presence of an alternating current source will generate the electromagnetic

waves in this domain which illuminate the target medium Ω.

We make certain assumptions about the material which are reflected in the constitutive relations in the domain Ω. For the biological media of interest to us, we can neglect magnetic effects; we also assume that Ohm's law governs the electric conductivity. Hence for $\vec{x} \in \Omega$

$$\begin{aligned} \vec{M}(\vec{x}) &= 0 \\ \vec{J}_c(\vec{x}) &= \sigma \vec{E}(\vec{x}). \end{aligned}$$

The parameter σ is among those which need to be identified to determine the dielectric properties of the media. We note that for biomedical applications, the assumption of Ohm's law may not be appropriate (e.g., see [APM]) and one might need to also include a hysteretic dependency (in terms of a conductivity susceptibility kernel similar to the polarization kernel in (2.1) below). While such a more general conductivity relationship would add to the computational requirements, the ideas and techniques presented in this paper could readily be used to treat such an assumption. For simplicity we restrict ourselves to Ohmic conductivity in our discussion here.

To describe the behavior of the media's electric polarization, we employ a general integral equation model in which the polarization explicitly depends on the past history of the electric field. The resulting constitutive law can be given in terms of a *polarization or displacement susceptibility kernel* g by

$$\vec{P}(t, \vec{x}) = \int_0^t g(t - s, \vec{x}) \vec{E}(s, \vec{x}) ds. \tag{2.1}$$

We note that this model presupposes that $\vec{P}(0, \vec{x}) = 0$. In the electromagnetic literature, e.g., see [Jac, BK, APM], the relationship is often expressed as

$$\vec{P}(t, \vec{x}) = \int_0^\infty G(\xi, \vec{x}) \vec{E}(t - \xi, \vec{x}) d\xi$$

or as

$$\vec{P}(t, \vec{x}) = \int_0^t G(\xi, \vec{x}) \vec{E}(t - \xi, \vec{x}) d\xi$$

in the case of $\vec{E}(t,\vec{x}) = 0$ for $t < 0$ which is of interest here. These are both related to our formulation by the simple change of variables: $s = t - \xi$. We prefer our form of the equation since then any time derivatives are borne by the kernel function g and not the variable E. Specifically, under (2.1) the term $\dfrac{\partial^2 \vec{P}}{\partial t^2}(t,\vec{x}) = \ddot{\vec{P}}(t,\vec{x})$, which will appear in subsequent equations, is given by

$$\ddot{\vec{P}}(t,\vec{x}) = \int_0^t \ddot{g}(t-s,\vec{x})\vec{E}(s,\vec{x})ds + g(0,\vec{x})\dot{\vec{E}}(t,\vec{x}) + \dot{g}(0,\vec{x})\vec{E}(t,\vec{x}) \tag{2.2}$$

while the more traditional representation leads to

$$\ddot{\vec{P}}(t,\vec{x}) = \int_0^t G(\xi,\vec{x})\ddot{\vec{E}}(t-\xi,\vec{x})ds + G(t,\vec{x})\vec{E}(0,\vec{x}) + \dot{G}(t,\vec{x})\vec{E}(0,\vec{x}). \tag{2.3}$$

The presence of $\ddot{\vec{E}}$ under the integral term in equation (2.3) complicates the analysis and solution of the Maxwell's equations considerably. Although the first formulation leads to the additional terms $g(0,\vec{x})\dot{\vec{E}}(t,\vec{x})$ and $\dot{g}(0,\vec{x})\dot{\vec{E}}(t,\vec{x})$, as we shall see below, these terms cause no increase in the complexity of the problem analysis or computation.

We note that an attempt to include a component of the polarization which depends on the instantaneous value of the electric field would add a delta function in the time variable to the kernel function $g(s,\vec{x})$. This introduces some mathematical complexities which, for simplicity, we avoid by treating instantaneous polarization when it arises in a different, but completely equivalent, manner (i.e., by modifying the first term in the displacement/electric field relationship). The equation relating the electric flux density to the electric field can be readily replaced by

$$\vec{D} = \epsilon_r \epsilon_0 \vec{E} + \vec{P}$$

where $\epsilon_r \geq 1$ is defined as the relative permittivity. This formulation, which is standard in the electromagnetic literature, introduces another parameter ϵ_r which can be treated as a spatially dependent parameter to allow for instantaneous effects on displacement in Ω due to the electric field originating in Ω_0.

The constitutive law in equation (2.1) is also sufficiently general to include models based on differential equations and systems of differential equations, or delay differential equations, (see [BJ]) whose solutions can be expressed through fundamental solutions (in general variation-of-parameters representations). For example, the choice of kernel function $g(t) = e^{-t\tau}(\epsilon_s - \epsilon_\infty)/\tau$ in Ω corresponds to the differential equation of the Debye model in Ω given by

$$\begin{aligned} \tau\dot{\vec{P}} + \vec{P} &= \epsilon_0(\epsilon_s - \epsilon_\infty)\vec{E} \\ \vec{D} &= \epsilon_\infty\epsilon_0\vec{E} + \vec{P}. \end{aligned} \tag{2.4}$$

The presence of instantaneous polarization is accounted for in this case by the coefficient ϵ_∞ in the electric flux equation. That is, $\epsilon_r = \epsilon_\infty$ in $\Omega, \epsilon_r = 1$ in Ω_0. The remainder of the electric polarization is seen to be a decaying exponential, driven by the electric field, less the part included in the instantaneous polarization. This model was first proposed by Debye in [Deb, VH] to model the behavior of materials whose molecules have permanent dipole moments. The magnitude of the polarization term P represents the degree of alignment of these individual moments.

We will also consider the Lorentz model of electric polarization which, in differential form, is represented with the second order equation:

$$\begin{aligned} &\ddot{\vec{P}} + \frac{1}{\tau}\dot{\vec{P}} + \omega_0^2\vec{P} = \epsilon_0\omega_p^2\vec{E} \\ &\vec{D} = \epsilon_\infty\epsilon_0\vec{E} + \vec{P}. \end{aligned} \tag{2.5}$$

The so-called plasma frequency is defined to be $\omega_p = \omega_0\sqrt{\epsilon_s - \epsilon_\infty}$. A simple variation of constants solution yields the correct kernel function

$$g(t) = \frac{\epsilon_0\omega_p^2}{\nu_0}e^{-\frac{1}{2\tau}t}\sin(\nu_0 t)$$

where $\nu_0 = \sqrt{\omega_0^2 - \frac{1}{4\tau^2}}$.

2.2 Estimation Methodology

Adapting a rather standard approach, we propose to identify the unknown parameters in a given model of polarization and a geometric representation by attempting to minimize the difference between simulations and observations of time-domain data. The data are measurements of the electric field at points in the exterior domain Ω_0 at discrete times. The simulation is a computed solution to Maxwell's Equations with the constitutive laws for polarization, using candidate values of the geometric and material parameters. The criterion for optimization is the minimization of a least-squares measurement of the difference between the simulation and the observed data given by

$$J(\vec{q}) = \sum_{i=1}^{N} |\vec{E}(t_i, \vec{x}_i; \vec{q}) - \hat{E}_i|^2.$$

The $\hat{E}_i$ are measurements of the electric field taken at specific locations and times. The $\vec{E}(t_i, \vec{x}_i; \vec{q})$ are solutions evaluated at the same locations and times from the simulation using the full set of parameter values $\vec{q}$.

We note two nontrivial difficulties with this approach and propose solutions, the efficiency of which will be demonstrated in the particular implementation discussed in this paper.

The unknown location of part of the boundary creates computational challenges. During the course of an iterative optimization procedure, simulations will be repeated many times for different locations of the unknown part of the boundary of Ω. That is, iterative based methods generally will involve changing domains and hence changing discretization grids in the usual finite element or finite difference approximation schemes. Any associated computational scheme (with domain changing with each iterative step) will be prohibitive in effort and time.

We address this difficulty by employing the "method of mappings", [BKo, Pi, BKoW] and transforming the problem on $\Omega \bigcup \Omega_0$ with unknown geometry to one with known geometry (a reference domain $\tilde{\Omega}$) at the expense of introducing additional unknowns into the equations that must be solved on this new domain. Generally, we

cannot expect these mappings to be C^2, or even C^1, which precludes finding classical solutions of the transformed system. For this and other reasons discussed below, we consider a weak formulation of the problem. The effect of the mapping is seen in additional coefficients appearing in the equations. The values of these coefficients must be interpreted through the inverse map to determine the geometric parameters.

Another difficulty arises from the oscillatory nature of the time domain data. Varying some parameters in the model has the effect of changing the time at which reflected signals are detected by varying the distance the propagated waves travel (to and from the unknown boundary $\Gamma(q)$) and their speed of propagation. This causes the simulated data to move in and out of phase with observations as these parameters are changed, producing in the cost function J an oscillatory character with respect to these parameters. Thus, local minima of J can (and do) arise at values for which the simulation moves partially back into phase with the data (this is illustrated in Section 4).

We choose to avoid the resulting difficult optimization problem by employing a multi-step approach which separates the identification of geometric and physical parameters. (Other approaches are possible, e.g., use of an optimization algorithm that obtains global minima even in the presence of multiple local minima.) The data set is truncated in time to a period which contains only partial reflections from initial penetrations of the interrogating signal on the surface of the material, eliminating the dependence on geometry. This generates an estimate of the dielectric parameters which is used in a second step which attempts to recover the geometry. To avoid the oscillatory nature of the objective function, a different optimization criterion is used in the second step. This criterion compares the secondary return times (i.e., return times after the pulse has reflected back from the unknown supraconductive-backed part of the boundary) of pulsed signals through mediums of varying geometry with the observed data secondary return times. A global optimization for improved estimates of all parameters is then attempted in a third step, using the estimates of the geometry and the prior estimate of the dielectric parameters as initial estimates.

2.3 Reduction to a Specific Problem

The choice of an interrogating input signal (in our case, a windowed microwave pulse from an exterior antenna in Ω_0) has profound implications on both theoretical and computational aspects of the inverse problem for estimation of dielectric and geometric parameters. A very popular (and readily implemented) choice consists of a *polarized* planar wave. This produces a signal with the $\vec{E}$ and $\vec{H}$ fields possessing nontrivial components in only one dimension in Ω_0. If the interrogated medium Ω has some homogeneity (in planes parallel to that of the interrogating planar wave), a similar reduction of the $\vec{E}$ and $\vec{H}$ fields occurs in the body Ω.

To discuss our inverse problem ideas, we consider in this paper the problem of interrogating an infinite slab of homogeneous (in the directions orthogonal to the direction of propagation of the plane wave) material by a polarized plane wave windowed microwave pulse. Specifically, as depicted in Figure 2.2, the interrogating signal is assumed to be a planar electro-magnetic wave normally incident on an infinite slab of material contained in the interval $[z_1, z_2]$ with faces parallel to the xy plane. The electric field is polarized with oscillations in the xz plane only.

Under these assumptions, it is easy to argue that the electric field is parallel to the $\hat{\imath}$ axis at all points in Ω_0 (the region external to the slab) and that the magnetic field is always parallel to $\hat{\jmath}$. Furthermore, these fields are homogeneous in intensity in the x and y directions. Thus $\vec{E}(t,\vec{x}) = \hat{\imath}E(t,z), \vec{H}(t,\vec{x}) = \hat{\jmath}H(t,z)$ as shown in Figure 2.2. The electric flux density $\vec{D}$ and polarization $\vec{P}$ inherit this uniform directional property from $\vec{E}$ and hence will be denoted hereafter by their scalar magnitudes D and P in the $\hat{\imath}$ direction. Since we have assumed that the material properties are homogeneous in the x and y variables, the propagating waves in Ω are also reduced to one nontrivial component. Thus the problem's dependence on x and y disappears since the resulting fields are necessarily homogeneous in these variables. This makes it possible to represent the fields in Ω and Ω_0 with the scalar functions $E(t,z)$ and $H(t,z)$. Under these assumptions, the differential operation $\nabla \times \vec{A}$ reduces to $-\hat{\imath}\frac{\partial A_y}{\partial z} + \hat{\jmath}\frac{\partial A_x}{\partial z}$

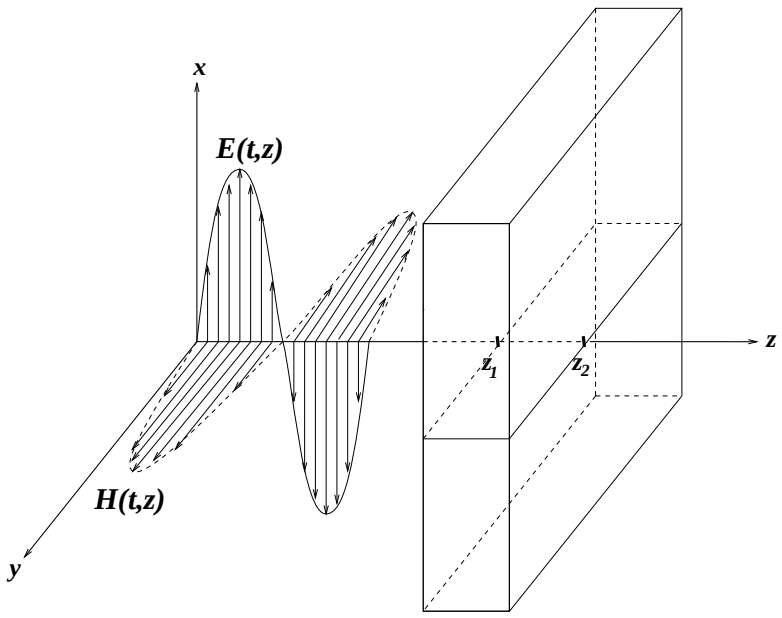

Figure 2.2: Geometry of Physical Problem

and the Maxwell's Equations from Section 2.1 become

$$\frac{\partial E}{\partial z} = -\mu_o \frac{\partial H}{\partial t} \tag{2.6}$$

$$-\frac{\partial H}{\partial z} = \frac{\partial D}{\partial t} + \sigma E + J_s. \tag{2.7}$$

We eliminate the magnetic field from the equations by taking the space derivative of equation (2.6), the time derivative of equation (2.7) and using the equation for electric flux density $D = \epsilon E + P$ where $\epsilon = \epsilon_0(1 + (\epsilon_r - 1)I_\Omega)$ to obtain

$$\mu_0 \epsilon \ddot{E} + \mu_0 \ddot{P} + \mu_0 \sigma \dot{E} - E'' = -\mu_0 \dot{J}_s. \tag{2.8}$$

Here and throughout, I_A will denote the indicator function for a set A and $' = \frac{\partial}{\partial z}, \dot{} = \frac{\partial}{\partial t}$.

This is the differential equation of concern for both our simulations and inverse problem calculations. We also define the domain of the computation (after the method of mappings has been applied) to be the interval $\tilde{\Omega} = [0, 1]$ which contains Ω_0. An

absorbing boundary condition is placed at the $z = 0$ boundary of the interval to prevent the reflection of waves. This can be expressed by

$$\left[\frac{1}{c}\frac{\partial E}{\partial t} - \frac{\partial E}{\partial z}\right]_{z=0} = 0 \tag{2.9}$$

where $c^2 \equiv 1/\epsilon_0\mu_0$.

For our formulation we assume that the location of the boundary at $z = z_1$ is known, while the location of the *original* back boundary at $z = z_2$ (equivalently, the depth of the slab) is unknown, i.e., $\Gamma(q) = \{\vec{x} \in \mathbb{R}^3 | z = z_2\}$. Thus we have a supraconductive backing on the slab at $z = z_2$. The boundary conditions on this supra-conductive reflector can now be determined explicitly, again assuming that the method of maps had been applied and therefore the back boundary of the material is now found at $z = 1$, the edge of the computational domain. (The precise map will be discussed momentarily below.) The vector normal to the surface is $\hat{n} = \hat{k}$ and the condition $\vec{B} \cdot \hat{n}$ is satisfied automatically since $\vec{B} = B\hat{\jmath}$. The condition $\vec{E} \times \hat{n} = 0$ becomes $[E_y\hat{\imath} - E_x\hat{\jmath}]_\Gamma = 0$ and hence the condition $E_x = E_y = 0$. Since $\vec{E} = (E_x, 0, 0) = \hat{\imath}E$ this the equivalent to the condition that E(t,1)=0.

Substituting the expression for $\ddot{P}$ derived in equation (2.2) we obtain the strong form of the equation

$$\begin{aligned} &\tilde{\epsilon}_r\ddot{E}(t,z) + \frac{1}{\epsilon_0}I_\Omega(z)(\sigma(z) + g(0,z))\dot{E}(t,z) \\ &\quad + \frac{1}{\epsilon_0}I_\Omega\dot{g}(0,z)E(t,z) + \int_0^t I_\Omega\frac{1}{\epsilon_0}\ddot{g}(t-s,z)E(s,z)ds \\ &\quad -c^2E''(t,z) = -\frac{1}{\epsilon_0}J_t(t,z), \end{aligned} \tag{2.10}$$

where indicator functions I_Ω have been added to explicitly enforce the restriction of polarization and conductivity to the interior of the transformed medium $\Omega = [z_1, 1]$ and $\tilde{\epsilon}_r = \epsilon/\epsilon_0 = 1 + (\epsilon_r - 1)I_\Omega \geq 1$ throughout $\Omega_0 \bigcup \Omega$.

Alternatively, we can apply integration by parts to the integral term and arrive at a different form of the equation.

$$\tilde{\epsilon}_r\ddot{E}(t,x) + \frac{1}{\epsilon_0}I_\Omega(z)(\sigma(z) + g(0,z))\dot{E}(t,z)$$

$$+\frac{1}{\epsilon_0}I_\Omega \dot{g}(t,z)E(0,z) + \int_0^t \frac{1}{\epsilon_0}I_\Omega \dot{g}(t-s,z)\dot{E}(s,z)ds \tag{2.11}$$
$$-c^2E''(t,z) = -\frac{1}{\epsilon_0}\dot{J}_s(t,z).$$

This form of the equation has the advantage of requiring less continuity in the hysteresis function $g(s,z)$. Furthermore, one of its terms disappears when $E(0,z)=0$, which is the case of interest in the simulations. We shall use the weak form of this equation in our computational investigations in Section 4.5. Analysis of the well posedness of equation (2.10) in weak form is carried out in [BB].

Due to the forms of the interrogating inputs, the dielectrically discontinuous medium interfaces, and the possible lack of smoothness in mapping the original domain $\Omega_0 \bigcup \Omega = [0, z_2]$ to the reference domain $\tilde{\Omega} = [0,1]$, one should not expect classical solutions to Maxwell's equations in strong form. For both theoretical and computational purposes, it is therefore desirable to write the system equations in weak or variational form. Using the spaces $H = L_2(0,1)$ and $V = H^1_R(0,1) = \{\phi \in H^1(0,1) | \phi(1) = 0\}$, we can express equation (2.11) in the more general weak form as

$$\langle \tilde{\epsilon}_r \ddot{E}, \phi\rangle + \langle \gamma \dot{E}, \phi\rangle + \langle \beta E, \phi\rangle + \langle \int_0^t \alpha(t-s,\cdot)\dot{E}(s)ds, \phi\rangle$$
$$+\kappa(t,z)E(0,z) + \langle c^2E', \phi'\rangle + cE_t(t,0)\phi(0) = \langle \mathcal{J}(t,\cdot), \phi\rangle \tag{2.12}$$

with initial conditions

$$E(0,z) = \Phi(z) \quad \dot{E}(0,z) = \Psi(z)$$

The coefficients of our particular problem in (2.12) are given by

$$\alpha(t,z) = \tfrac{1}{\epsilon_0}I_\Omega(z)\dot{g}(t,z), \qquad \beta(z) = 0,$$
$$\gamma(z) = \tfrac{1}{\epsilon_0}I_\Omega(z)(\sigma(z) + g(0,z)), \quad \kappa(t,z) = -\tfrac{1}{\epsilon_0}I_\Omega(z)\dot{g}(t,z),$$
$$\mathcal{J}(t,z) = -\tfrac{1}{\epsilon_0}\dot{J}(t,z)$$

and $\langle \cdot, \cdot \rangle$ is the L^2 inner product (or equivalently, any appropriately chosen topologically equivalent inner product - the relevance of this remark will be clearer after our discussion below of the method of maps for this example). We retain the coefficient

β in the analysis to address the more general problem posed by non-zero functions in this coefficient. The functions β, γ and κ are assumed to be in L^∞ but may lack any additional regularity. For the purpose of demonstrating uniqueness of solutions to equation 2.12 we will require $\alpha \in H^{1,\infty}$, that is, both $\alpha \in L^\infty$ and $\dot{\alpha} \in L^\infty$.

The source current is under our control, so we can and do choose its form precisely. For example, we may use

$$J_s(t, z) = \delta(z) \sin(\omega t) I_{(0,t_f)}(t). \tag{2.13}$$

Here, ω is a specified angular frequency of the input signal (and the carrier frequency of the resulting planar wave) and $\delta(z)$ is the Dirac distribution which has infinite mass at $z = 0$. The signal is truncated at a finite time t_f by the indicator function $I_{(0,t_f)}(t)$. We avoid the complications arising from a discontinuous (in t) input signal by requiring that the sinusoid be zero at t_f. Hence $\omega t_f = n\pi$ for some positive integer n. This is equivalent to requiring that the end of the signal occur after a integral number of half-periods. We note that this windowed signal can be equivalently chosen with additional smoothness in t by replacing the indicator $I_{(0,t_f)}$ with a slightly smoother (continuous or even differentiable) truncating function.

This windowed pulse input signal is most helpful in identifying the physical and geometric parameters separately. Since it has a finite duration, the wave's reflection off the surface of the media and its subsequent reflection off the back surface will not necessarily overlap for a sufficiently short pulse. This makes it possible to split the resulting data and perform the two estimation steps described above separately.

We turn to the details of the method of maps for this example. The application of the method of mappings is quite straightforward in this particular case. The value of z_1 is presumed known, while the surface at z_2 is inaccessible and therefore its location (i.e., the value of z_2) is not known. We use a piece-wise linear mapping which leaves the interval $(0, z_1)$ invariant and maps (z_1, z_2) to $(z_1, 1)$, thus mapping the original domain $\Omega_0 \bigcup \Omega = [0, z_2]$ to the reference domain $\tilde{\Omega} = [0, 1]$. The new coordinate

variable $\tilde{z}$ in $\tilde{\Omega} = [0,1]$ is defined by

$$\tilde{z} = f(z) = \begin{cases} z & 0 < z < z_1 \\ z_1 + (z - z_1)\frac{1-z_1}{z_2-z_1} & z_1 \leq z \leq z_2 \end{cases}$$

We can express the function as

$$f(z) = z + (\zeta - 1)(z - z_1)I_{[z_1,z_2]}(z)$$

where $\zeta = \frac{1-z_1}{z_2-z_1}$ and

$$f'(z) = 1 + (\zeta - 1)I_{\Omega}(z).$$

The parameter ζ appearing the equations can be identified in lieu of the depth $z_2 - z_1$ which of course can be recovered from it. Wherever spatial derivatives appear in the equations, we must replace these with the derivatives with respect to the new variable $\tilde{z}$ using the chain rule

$$\frac{\partial}{\partial z} = \frac{\partial \tilde{z}}{\partial z}\frac{\partial}{\partial \tilde{z}} = f'(z)\frac{\partial}{\partial \tilde{z}}.$$

Hence $d\tilde{z} = f'(z)dz$ and the expressions for the inner products in the weak form are modified accordingly to

$$\begin{aligned} \langle \phi, \psi \rangle &= \int_0^{z_2} \phi(z)\psi(z)dz \\ &= \int_0^1 \tilde{\phi}(\tilde{z})\tilde{\psi}(\tilde{z})\frac{d\tilde{z}}{\tilde{f}'(\tilde{z})} \\ &= \int_0^1 \frac{\tilde{\phi}(\tilde{z})\tilde{\psi}(\tilde{z})}{1 + (\zeta - 1)I_{\Omega}}d\tilde{z} \\ &= \int_0^{z_1} \tilde{\phi}(\tilde{z})\tilde{\psi}(\tilde{z})d\tilde{z} + \frac{1}{\zeta}\int_{z_1}^1 \tilde{\phi}(\tilde{z})\tilde{\psi}(\tilde{z})d\tilde{z} \end{aligned} \tag{2.14}$$

$$\begin{aligned} \langle \phi', \psi' \rangle &= \int_0^{z_2} \phi'(z)\psi'(z)dz \\ &= \int_0^1 \tilde{\phi}'(\tilde{z})\tilde{f}'(\tilde{z})\tilde{\psi}'(\tilde{z})\tilde{f}'(\tilde{z})\frac{d\tilde{z}}{\tilde{f}'(\tilde{z})} \\ &= \int_0^1 \tilde{\phi}'(\tilde{z})\tilde{\psi}'(\tilde{z})\tilde{f}'(\tilde{z})d\tilde{z} \\ &= \int_0^1 (1 + (\zeta - 1)I_{\Omega})\tilde{\phi}'(\tilde{z})\tilde{\psi}'(\tilde{z})d\tilde{z} \end{aligned}$$

$$= \int_0^{z_1} \tilde{\phi}'(\tilde{z})\tilde{\psi}'(\tilde{z})d\tilde{z} + \zeta \int_{z_1}^1 \tilde{\phi}'(\tilde{z})\tilde{\psi}'(\tilde{z})d\tilde{z}. \tag{2.15}$$

Here the functions ϕ and ψ have been expressed in terms of the new variables $\phi(z) = \tilde{\phi}(\tilde{z}), \psi(z) = \tilde{\psi}(\tilde{z})$. For convenience, we will drop the over tilde notation in subsequent discussion.

Returning to equation (2.12) again, we see that in mapping the original Maxwell system on the domain $\Omega_0 \bigcup \Omega$ to the reference domain $\tilde{\Omega} = [0, 1]$, the $\langle \cdot, \cdot \rangle$ in (2.12) should be interpreted in the sense (2.14),(2.15) above. This will be done in all computational results presented in Section 4. Note that for the theoretical discussions of Section 3 we can without loss of generality use the usual L^2 inner products.

This one dimensional problem also permits a simple interpretation of the return time when identifying the geometry. Before the method of maps is applied, the unknown geometric quantity is the depth, $z_2 - z_1$, of the medium. After the mapping is applied, the actual depth of the medium is fixed, but the effects of changing geometry are reflected in the new coefficients, (actually the weighting parameter ζ in the inner products) which appears in the equation. Since there is only one path for the pulse to pass through the medium and be reflected back, the first return of the transmitted signal is a singular event and simple to identify.

To further clarify the identification of the return time, we restrict our attention to materials which are sufficiently thick and pulses which are sufficiently short that the first reflection of the pulse off the surface at z_1 is completed before the transmitted signal has returned from the back surface. Using c, the speed of light in a vacuum, as an upper limit on the speed of propagation in the material, this condition requires that $t_f c < 2(z_2 - z_1)$.

Chapter 3

Wellposedness

In this section we consider wellposedness questions for the variational form (2.12) of Maxwell's equation with a general polarization term, absorbing left boundary conditions and supra-conductive right boundary conditions as discussed and formulated in the previous section. Through the use of a topologically equivalent definition of the inner product, we may take $\tilde{\epsilon}_r = 1$ for our discussion in this section. That is, we are concerned with existence, uniqueness, continuous dependence and regularity of solutions to

$$\begin{aligned} \langle \ddot{E}, \phi \rangle + \langle \gamma \dot{E}, \phi \rangle + \langle \beta E, \phi \rangle + \langle \int_0^t \alpha(t-s)\dot{E}(s)ds, \phi \rangle + \\ \langle \kappa(t)\Phi, \phi \rangle + \langle c^2 E', \phi' \rangle + c\dot{E}(t,0)\phi(0) = \langle \mathcal{J}(t), \phi \rangle \qquad (3.1) \\ E(0,z) = \Phi(z), \dot{E}(0,z) = \Psi(z). \qquad (3.2) \end{aligned}$$

We seek solutions $t \to E(t)$ with $E(t)$ in $V \equiv H_R^1(0,1) = \{\phi \in H^1(0,1) : \phi(1) = 0\}$ satisfying (3.2) and (3.1) for all $\phi \in V$. We shall do this in the context of a Gelfand triple setting $V \hookrightarrow H \hookrightarrow V^*$ where $H = L^2(0,1)$. We assume throughout this section that the slab region $\Omega = [z_1, z_2]$ has been mapped to $[z_1, 1]$ so that our domain of interest is $\tilde{\Omega} = [0,1]$ with $\gamma, \beta, \alpha(t)$ and $\kappa(t)$ bounded on $[0,1]$ and vanishing outside $[z_1, 1]$. Thus, in (3.1) the inner products should be properly interpreted in the sense of (2.14),(2.15). However, as explained in the last section, one can equivalently treat the well-posedness questions of this section using the unweighted L^2 inner product

in (3.1) and we shall do this.

From a general theory presented in [BIW], [BSW, Chapter 4], one sees that (3.1) differs from the usual lightly damped second order systems of [BSW] only because of the presence of the terms $c\dot{E}(t,0)\phi(0)$ due to the absorbing left boundary condition and $\langle \int_0^t \alpha(t-s)\dot{E}(s)ds, \phi\rangle + \langle \kappa(t)E(0), \phi\rangle$ resulting from the convolution representation for the polarization. From the general theory one might expect to seek solutions of (3.1) in the sense of $L^2(0,T;V)^* \simeq L^2(0,T;V^*)$ with $E \in L^2(0,T;V), \dot{E} \in L^2(0,T;H)$ and $\ddot{E} \in L^2(0,T;V^*)$ for appropriate interpretation of the $\langle\cdot,\cdot\rangle$ in (3.1) i.e., the duality product $\langle\cdot,\cdot\rangle_{V^*,V}$ which reduces to the $H = L^2$ inner product in all terms of (3.1) except the first and last. We recall from Section 2.3 that the input $\mathcal{J}(t)$ results from a point source (antenna) at $z = 0$ and hence $\mathcal{J}(t)$ has the form $g(t)\delta(z)$ for a windowed time signal $g(t)$. This motivates our desire for results allowing $\mathcal{J}(t)$ values in V^*.

In addition to differences one might encounter due to the polarization term, if one obtains as usual $\dot{E} \in L^2(0,T;H)$ where $H = L^2(0,1)$, then questions arise as to the interpretation of the boundary term $c\dot{E}(t,0)\phi(0)$ which has the appearance of pointwise evaluation of an $L^2(0,1)$ "function" at $z = 0$. Correct interpretation of this term will result from our arguments below.

We follow the general approach using sesquilinear forms as in [BIW],[BSW] which are standard in the research literature [Lions], [Wloka]. We first rewrite (3.1) by adding a term $\langle kE, \phi\rangle_H$ to both sides of the equation. The positive constant k is chosen so that $\hat{\beta} \equiv k+\beta$ satisfies $\hat{\beta} \geq \hat{\epsilon} > 0$ on $[0,1]$ for some constant $\hat{\epsilon}$; this is possible since by assumption $\beta \in L^\infty(0,1)$. We define a sesquilinear form $\sigma_1 : V \times V \longrightarrow \mathbb{C}$ by

$$\sigma_1(\phi,\psi) = \langle c^2\phi', \psi'\rangle_H + \langle \hat{\beta}\phi, \psi\rangle_H \quad \text{for } \phi, \psi \in V.$$

Equation (3.1) can then be rewritten as

$$\langle \ddot{E}(t), \phi\rangle_{V^*,V} + \langle \gamma\dot{E}(t), \phi\rangle + \langle \int_0^t \alpha(t-s)\dot{E}(s)ds, \phi\rangle + \sigma_1(E(t),\phi) + \langle \kappa(t)\Phi, \phi\rangle + c\dot{E}(t,0)\phi(0) = \langle \mathcal{J}(t), \phi\rangle_{V^*,V} + \langle kE(t), \phi\rangle \tag{3.3}$$

for all $\phi \in V$, where it is readily seen that σ_1 is V-continuous and V-elliptic. That is, there are positive constants c_1, c_2 such that

$$\begin{aligned} |\sigma_1(\phi,\psi)| &\leq c_2|\phi|_V|\psi|_V \quad \text{for all } \phi,\psi \in V, \\ \sigma_1(\phi,\phi) &\geq c_1|\phi|_V^2 \quad \text{for } \phi \in V. \end{aligned}$$

To establish existence of solutions (3.2), (3.3) where $\Phi \in V, \Psi \in H$, we follow the ideas in [BSW] and choose a subset $\{w_i\}_{i=1}^{\infty}$ spanning V (without loss of generality we may assume linear independence of these elements). Let $V^m \equiv \text{span}\{w_1, \ldots, w_m\}$ and define Galerkin "approximates"

$$E_m(t,z) = \sum_{i=1}^{m} e_i^m(t) w_i(z) \tag{3.4}$$

where the $\{e_i^m(t)\}_{i=1}^m$ are determined by substitution into (3.3) and requiring this system of ordinary differential equations to hold for $\phi = w_i, i = 1, 2, \ldots, m$. This m-dimensional system is solved with initial conditions

$$\begin{aligned} E_m(0) &= \Phi_m \\ \dot{E}_m(0) &= \Psi_m \end{aligned}$$

where the Φ_m, Ψ_m are chosen in V^m so that $\Phi_m \to \Phi$ in V, $\Psi_m \to \Psi$ in H. We thus find that $E_m(t)$ satisfies (3.3) with $\phi = \dot{E}_m(t) \in V$ so that for each t we have the system

$$\begin{aligned} &\frac{d}{dt}\frac{1}{2}|\dot{E}_m(t)|_H^2 + \langle \gamma \dot{E}_m(t), \dot{E}_m(t)\rangle_H + \langle \kappa(t)\Phi, \dot{E}_m(t)\rangle_H \\ &\quad + \langle \int_0^t \alpha(t-s)\dot{E}_m(s)ds, \dot{E}_m(t)\rangle_H + \sigma_1(E_m(t), \dot{E}_m(t)) + c|\dot{E}_m(t,0)|^2 \\ &\quad = \langle \mathcal{J}(t), \dot{E}_m(t)\rangle_{V^*,V} + \langle kE_m(t), \dot{E}_m(t)\rangle_H, \qquad (3.5) \\ &E_m(0) = \Phi_m, \dot{E}_m(0) = \Psi_m. \end{aligned}$$

This system will allow us to obtain bounds for $\{E_m\}, \{\dot{E}_m\}$, and $\{\dot{E}_m(\cdot,0)\}$ that are independent of m.

Using the fact that

$$\frac{1}{2}\frac{d}{dt}\sigma_1(E_m(t),E_m(t)) = \sigma_1(E_m(t),\dot{E}_m(t)),$$

we may rewrite (3.5) as

$$\begin{aligned}&\frac{1}{2}\frac{d}{dt}\{|\dot{E}_m(t)|_H^2 + \sigma_1(E_m(t),E_m(t))\} + |\sqrt{\gamma}\dot{E}_m(t)|_H^2 + c|\dot{E}_m(t,0)|^2\\ &\quad = \langle -\int_0^t \alpha(t-s)E_m(s)ds, \dot{E}_m(t)\rangle_H + \langle kE_m(t),\dot{E}_m(t)\rangle_H + \langle -\kappa\Phi,\dot{E}_m(t)\rangle\\ &\quad + \langle \mathcal{J}(t),\dot{E}_m(t)\rangle_{V^*,V}.\end{aligned}$$

Integration along with use of the V ellipticity of σ_1 yields

$$\begin{aligned}|\dot{E}_m(t)|_H^2 + c_1|E_m(t)|_V^2 + 2\int_0^t |\sqrt{\gamma}\dot{E}_m(s)|_H^2 ds + 2c|\dot{E}_m(\cdot,0)|^2_{L^2(0,t)}&\\ \leq |\dot{E}_m(0)|_H^2 + c_2|E_m(0)|_V^2 + 2|\int_0^t F_m(\tau)d\tau|&\end{aligned} \tag{3.6}$$

where

$$\begin{aligned}F_m(\tau) &= \langle -\int_0^\tau \alpha(\tau-s)\dot{E}_m(s)ds, \dot{E}_m(\tau)\rangle + \langle kE_m(\tau),\dot{E}_m(\tau)\rangle + \langle \kappa\Phi,\dot{E}_m(\tau)\rangle\\ &\quad + \langle \mathcal{J}(\tau),\dot{E}_m(\tau)\rangle_{V^*,V}\\ &\equiv T_1(\tau)+T_2(\tau)+T_3(\tau)+T_4(\tau).\end{aligned}$$

Assuming that α is bounded on $[0,T]\times[0,1]$ we have for $\tau\in[0,t]$

$$\begin{aligned}|T_1(\tau)| &\leq \int_0^\tau K|\dot{E}_m(s)|_H ds |\dot{E}_m(\tau)|_H\\ &\leq \frac{1}{2}(\int_0^\tau K|\dot{E}_m(s)|_H ds)^2 + \frac{1}{2}|\dot{E}_m(\tau)|_H^2\\ &\leq \frac{1}{2}K^2 t\int_0^\tau |\dot{E}_m(s)|_H^2 ds + \frac{1}{2}|\dot{E}_m(\tau)|_H^2\\ &\leq \frac{1}{2}K^2 t\int_0^t |\dot{E}_m(s)|_H^2 ds + \frac{1}{2}|\dot{E}_m(\tau)|_H^2.\end{aligned}$$

Thus we find

$$\int_0^t |T_1(\tau)|d\tau \leq K_1\int_0^t |\dot{E}_m(s)|_H^2 ds \tag{3.7}$$

We also have

$$\int_0^t |T_2(\tau)|d\tau \leq \int_0^t \{\frac{1}{2}k^2|E_m(\tau)|_H^2 + \frac{1}{2}|\dot{E}_m(\tau)|_H^2\}d\tau \tag{3.8}$$

and, assuming that $\kappa(t)$ is bounded on $[0,T]\times[0,1]$ also, we find

$$\int_0^t |T_3(\tau)|d\tau \leq t\frac{L^2}{2}|\Phi|_H^2 + \int_0^t \frac{1}{2}|\dot{E}_m(t)|_H^2 d\tau. \tag{3.9}$$

Finally to consider the term T_4, we use (assuming that $\dot{\mathcal{J}} \in L^2(0,T;V^*)$)

$$\frac{d}{dt}\langle \mathcal{J}(t), E_m(t)\rangle_{V^*,V} = \langle \dot{\mathcal{J}}(t), E_m(t)\rangle_{V^*,V} + \langle \mathcal{J}(t), \dot{E}_m(t)\rangle_{V^*,V}.$$

We obtain

$$\begin{aligned}
\Big|\int_0^t T_4(\tau)d\tau\Big| &= \Big|\int_0^t \{\frac{d}{d\tau}\langle \mathcal{J}(\tau), E_m(\tau)\rangle_{V^*,V} - \langle \dot{\mathcal{J}}(\tau), E_m(\tau)\rangle_{V^*,V}\}d\tau\Big| \\
&= \Big|\langle \mathcal{J}(t), E_m(t)\rangle_{V^*,V} - \langle \mathcal{J}(0), E_m(0)\rangle_{V^*,V} - \int_0^t \langle \dot{\mathcal{J}}(\tau), E_m(\tau)\rangle_{V^*,V} d\tau\Big| \\
&\leq \frac{1}{4\epsilon}|\mathcal{J}(t)|_{V^*}^2 + \epsilon|E_m(t)|_V^2 + \frac{1}{2}|\mathcal{J}(0)|_{V^*}^2 + \frac{1}{2}|E_m(0)|_V^2 \\
&\quad + \int_0^t \Big\{\frac{1}{2}|\dot{\mathcal{J}}(\tau)|_{V^*}^2 + \frac{1}{2}|E_m(\tau)|_V^2\Big\}d\tau.
\end{aligned} \tag{3.10}$$

Combining (3.6), (3.7), (3.8), (3.9), and (3.10) we obtain

$$\begin{aligned}
&|\dot{E}_m(t)|_H^2 + (c_1 - 2\epsilon)|E_m(t)|_V^2 + 2\int_0^t |\sqrt{\gamma}\dot{E}_m(s)|_H^2 ds + 2c|\dot{E}_m(\cdot,0)|_{L^2(0,t)}^2 \\
&\leq |\dot{E}_m(0)|_H^2 + (c_2+1)|E_m(0)|_V^2 + \frac{1}{2\epsilon}|\mathcal{J}(t)|_{V^*}^2 + |\mathcal{J}(0)|_{V^*}^2 + \int_0^t |\dot{\mathcal{J}}(s)|_{V^*}^2 ds \\
&+ C_1|\Phi|_H^2 + \int_0^t \{C_2|E_m(s)|_H^2 + C_3|\dot{E}_m(s)|_H^2 + C_4|E_m(s)|_V^2\}ds.
\end{aligned} \tag{3.11}$$

Assuming that $\mathcal{J} \in H^1(0,T;V^*)$ and using the boundedness of $\{E_m(0)\}$ in V and $\{\dot{E}_m(0)\}$ in H (which follows from the convergences of $\{\Phi_m\}$ and $\{\Psi_m\}$ respectively), we may employ Gronwall's inequality along with the inequality (3.11) to conclude that $\{\dot{E}_m\}$ is bounded in $C(0,T;H)$, $\{E_m\}$ is bounded in $C(0,T;V)$ and $\{\dot{E}_m(\cdot,0)\}$ is bounded in $L^2(0,T)$. Thus we find (extracting subsequences and reindexing as usual) there exist $E \in L^2(0,T;V)$, $\tilde{E} \in L^2(0,T;H)$ and $z_0 \in L^2(0,T)$ such that

$$E_m \to E \quad \text{weakly in} \quad L^2(0,T;V)$$

$$\dot{E}_m \to \tilde{E} \quad \text{weakly in} \quad L^2(0,T;H)$$
$$\dot{E}_m(\cdot,0) \to z_0 \quad \text{weakly in} \quad L^2(0,T).$$

The limit function E is a candidate for solution of (3.2), (3.3) and we must verify that $\tilde{E} = \dot{E}$, $z_0 = \dot{E}(\cdot,0)$ in some sense and that we may pass to the limit in the version of (3.3) for E_m to obtain (3.3) for the limit function. First we note that for each m

$$E_m(t) = E_m(0) + \int_0^t \dot{E}_m(s)ds \tag{3.12}$$

and

$$E_m(t,0) = E_m(0,0) + \int_0^t \dot{E}_m(s,0)ds. \tag{3.13}$$

Passing to the limit (in the weak H sense in (3.12)) we obtain

$$E(t) = \Phi + \int_0^t \tilde{E}(s)ds \tag{3.14}$$
$$E(t,0) = \Phi(0) + \int_0^t z_0(s)ds. \tag{3.15}$$

We find that (3.14) holds in the H sense for each $t \in [0,T]$ and hence $\dot{E} = \tilde{E}$ while (3.15) yields that $E(t,0)$ exists and is continuous in t. In actuality we have $E(t,0)$ is absolutely continuous with $\dot{E}(t,0) = z_0(t)$ for almost every t.

We note, in fact, that the same arguments used in [BSGII,Lemma 5.1(b)] can be used to establish that E_m also converges weakly in $C(0,T;H)$ to E so that $E \in C(0,T;H) \cap L^2(0,T;V)$.

Thus we have that our candidate E for solution of (3.2), (3.3) satisfies

$$E_m \to E \quad \text{weakly in} \quad L^2(0,T;V) \tag{3.16}$$
$$\dot{E}_m \to \dot{E} \quad \text{weakly in} \quad L^2(0,T;H) \tag{3.17}$$
$$\dot{E}_m(\cdot,0) \to \dot{E}(\cdot,0) \quad \text{weakly in} \quad L^2(0,T). \tag{3.18}$$

We must show that E satisfies (3.3). For this we follow directly the arguments of pp.100-101 in [BSW]. Taking $\psi \in C^1[0,T]$ with $\psi(T) = 0$ and choosing $\psi_j(t) \equiv \psi(t)w_j$

where the $\{w_j\}_{j=1}^{\infty}$ are as chosen before, we have, fixing j, that for all $m > j$, E_m must satisfy

$$\begin{aligned}\int_0^T \Big\{ &\langle \ddot{E}_m(t), \psi_j(t)\rangle + \langle \gamma \dot{E}_m(t), \psi_j(t)\rangle + \langle \int_0^t \alpha(t-s)\dot{E}_m(s)ds, \psi_j(t)\rangle \\ &+ \sigma_1(E_m(t), \psi_j(t)) + \langle \kappa(t)\Phi, \psi_j\rangle + c\dot{E}_m(t,0)\psi_j(t)(0)\Big\}dt \\ &= \int_0^T \Big\{ \langle \mathcal{J}(t), \psi_j(t)\rangle_{V^*,V} + \langle kE_m(t), \psi_j(t)\rangle \Big\}dt.\end{aligned}$$

Integrating by parts in the first term and then taking the limit as $m \to \infty$, with the convergences of (3.16)-(3.18) we obtain

$$\begin{aligned}\int_0^T \Big\{ &-\langle \dot{E}(t), \dot{\psi}_j(t)\rangle + \langle \gamma \dot{E}(t), \psi_j(t)\rangle + \langle \int_0^t \alpha(t-s)\dot{E}(s)ds, \psi_j(t)\rangle \\ &+ \sigma_1(E(t), \psi_j(t)) + \langle \kappa(t)\Phi, \psi_j\rangle + c\dot{E}(t,0)\psi_j(t)(0)\Big\}dt \\ &= \int_0^T \{\langle \mathcal{J}(t), \psi_j(t)\rangle_{V^*,V} + \langle kE(t), \psi_j(t)\rangle\}dt + \langle \Psi, \psi_j(0)\rangle.\end{aligned}$$

It follows that for every w_j we have in the $L^2(0,T)$ sense

$$\begin{aligned}&\frac{d}{dt}\langle \dot{E}(t), w_j\rangle + \langle \gamma \dot{E}(t), w_j\rangle + \langle \int_o^t \alpha(t-s)\dot{E}(s)ds, w_j\rangle \\ &+ \sigma_1(E(t), w_j) + \langle \kappa(t)\Phi, w_j\rangle + cE(t,0)w_j(0) = \langle \mathcal{J}(t), w_j\rangle_{V^*,V} + \langle kE(t), w_j\rangle.\end{aligned}$$

Since $\{w_j\}_{j=1}^{\infty}$ was chosen total in V we thus obtain that $\ddot{E} \in L^2(0,T;V^*)$ and that E satisfies (3.3). From (3.14) we know that $E(0) = \Phi$ and the arguments that $\dot{E} = \Psi$ follow exactly as those on p.101 of [BSW]. Hence we find that E is a solution of (3.2),(3.3).

Continuous dependence of solutions to (3.2), (3.3) on Φ, Ψ and $\mathcal{J}$ follow readily from the inequality (3.11) and some standard arguments. Noting that $|\cdot|_H \leq \mu|\cdot|_V$ for some constant μ and letting

$$\begin{aligned}K_m \equiv\ &|\dot{E}_m(0)|_H^2 + (c_2+1)|E_m(0)|_V^2 + \frac{1}{2\epsilon}|\mathcal{J}|^2_{L^\infty(0,T;V^*)} \\ &+ |\mathcal{J}(0)|^2_{V^*} + \int_0^T |\dot{\mathcal{J}}(s)|^2_{V^*}ds + C_1|\Psi|_H^2, \qquad (3.19)\end{aligned}$$

we observe that (3.11) implies

$$|\dot{E}_m(t)|_H^2 + |E_m(t)|_V^2 \leq \nu K_m + \int_0^t \nu\{|E_m(\tau)|_V^2 + |\dot{E}_m(\tau)|_H^2\}d\tau$$

for some positive constant ν independent of m. Using Gronwall's inequality again, we obtain

$$|\dot{E}_m(t)|_H^2 + |E_m(t)|_V^2 \leq \nu K_m e^{\nu T} \quad \text{for } t \in [0,T]. \tag{3.20}$$

Recalling that $E_m(0) = \Phi_m \to \Phi$ in V and $\dot{E}_m(0) = \Psi_m \to \Psi$ in H so that from (3.19) we have $\lim_{m\to\infty} K_m \leq K$ where $K \equiv |\Psi|_H^2 + (\mu C_1 + c_2 + 1)|\Phi|_V^2 + \mathcal{K}|\mathcal{J}|^2_{H^1(0,T;V^*)}$, we may use weak lower semicontinuity of norms, the convergences of (3.16) and (3.17), and (3.20) to conclude

$$|\dot{E}|^2_{L^2(0,T;H)} + |E|^2_{L^2(0,T;V)} \leq \nu K e^{\nu T} \tag{3.21}$$

Since the mapping $(\Phi, \Psi, \mathcal{J}) \to (E, \dot{E})$ is linear from $V \times H \times H^1(0,T;V^*)$ to $L^2(0,T;V) \times L^2(0,T;H)$ we see that (3.21) yields continuous dependence of solutions $(E, \dot{E})$ of (3.2), (3.3) on initial data (Φ, Ψ) and input $\mathcal{J}$.

For uniqueness of solutions to (3.2), (3.3), we again follow the standard arguments given in p.102-103 of [BSW]. In this case the details are tedious but rather straightforward. As usual, it suffices to show that the only solution of (3.3) corresponding to zero initial data ($\Phi = \Psi = 0$ in (3.2)) and zero input ($\mathcal{J} = 0$) is the trivial solution. Let E be a solution corresponding to $\Phi = \Psi = \mathcal{J} = 0$ and for arbitrary s in $(0,T)$ define

$$\psi_s(t) = \begin{cases} -\int_t^s E(\xi)d\xi & t < s \\ 0 & t \geq s \end{cases}$$

so that $\psi_s(T) = 0$ and $\psi_s(t) \in V$ for each t. We then find that

$$\int_0^s \{\langle \ddot{E}(t), \psi_s(t)\rangle_{V^*,V} + \langle \dot{E}(t), E(t)\rangle_H\}dt = 0.$$

Hence, choosing $\phi = \psi_s(t)$ in (3.3) and integrating over t from 0 to s, we have

$$\begin{aligned} \int_0^s \{&\langle \dot{E}(t), E(t)\rangle_H - \langle \gamma \dot{E}(t), \psi_s(t)\rangle_H - \langle \int_0^t \alpha(t-\tau)\dot{E}(\tau)d\tau, \psi_s(t)\rangle_H \\ &- \sigma_1(E(t), \psi_s(t)) - c\dot{E}(t,0)\psi_s(t)(0) + \langle kE(t), \psi_s(t)\rangle_H\}dt = 0. \end{aligned} \tag{3.22}$$

Observing that

$$\int_0^s \{\langle \gamma \dot{E}, \psi_s\rangle + \langle \gamma E, E\rangle\}dt = \int_0^s \frac{d}{dt}\langle \gamma E, \psi_s\rangle dt = 0,$$

$$\frac{d}{dt}\sigma_1(\psi_s(t),\psi_s(t)) = 2\mathrm{Re}\sigma_1(E(t),\psi_s(t)),$$

$$\int_0^s \{c\dot{E}(t,0)\psi_s(t)(0) + cE(t,0)^2\}dt = \int_0^s \frac{d}{dt}\{cE(t,0)\psi_s(t)(0)\}dt = 0$$

and

$$\frac{d}{dt}\langle k\psi_s(t),\psi_s(t)\rangle_H = 2\mathrm{Re}\langle kE(t),\psi_s(t)\rangle,$$

we may use (3.22) to obtain

$$\begin{aligned}\frac{1}{2}|E(s)|_H^2 &+ \frac{1}{2}\sigma_1(\psi_s(0)\psi_s(0)) + \frac{1}{2}\langle k\psi_s(0),\psi_s(0)\rangle \\ + &\int_0^s \{\langle \gamma E(t),E(t)\rangle + cE(t,0)^2 \\ &-\mathrm{Re}\langle \int_0^t \alpha(t-\tau)\dot{E}(\tau)d\tau,\psi_s(t)\rangle\}dt = 0. \end{aligned} \tag{3.23}$$

It follows immediately that

$$\begin{aligned}\frac{1}{2}|E(s)|_H^2 &\le \frac{1}{2}|\langle k\psi_s(0),\psi_s(0)\rangle| \\ + &\int_0^s \{|\langle \gamma E(t),E(t)\rangle| + |\langle \int_0^t \alpha(t-\tau)\dot{E}(\tau)d\tau,\psi_s(t)\rangle|\}dt. \end{aligned} \tag{3.24}$$

From the definition of ψ_s we have for each $t \in [0,T]$

$$|\psi_s(t)|_H^2 \le (\int_0^s |E(\xi)|_H d\xi)^2 \le T\int_0^s |E(\xi)|_H^2 d\xi \tag{3.25}$$

so that

$$\begin{aligned}|\langle k\psi_s(0),\psi_s(0)\rangle_H| &= |\int_0^1 k(\int_0^s E(\xi,z)d\xi)^2 dz| \\ &\le |\int_0^1 kT\int_0^s |E(\xi,z)|^2 d\xi dz| \\ &= kT\int_0^s |E(\xi)|_H^2 d\xi. \end{aligned} \tag{3.26}$$

Using (3.25) and arguments similar to those behind the estimate (3.7) for $T_1(\tau)$ we find for $t < s$

$$|\langle \int_0^t \alpha(t-\tau)\dot{E}(\tau)d\tau,\psi_s(t)\rangle| = |\langle -\int_0^t \dot{\alpha}(t-\tau)E(\tau)d\tau,\psi_s(t)\rangle|$$

$$\begin{aligned} &\leq (\int_0^t K_2|E(\tau)|_H d\tau)|\psi_s(t)|_H \\ &\leq \frac{1}{2}(\int_0^t K_2|E(\tau)|_H d\tau)^2 + \frac{1}{2}|\psi_s(t)|_H^2 \\ &\leq K_3 \int_0^s |E(\xi)|_H^2 d\xi. \end{aligned} \tag{3.27}$$

Using (3.26) and (3.27) we thus obtain

$$\frac{1}{2}|E(s)|_H^2 \leq \{\frac{1}{2}kT + |\gamma|_\infty + K_3\} \int_0^s |E(\xi)|_H^2 d\xi \tag{3.28}$$

or

$$|E(s)|_H^2 \leq \mathcal{K} \int_0^s |E(\xi)|_H^2 d\xi \tag{3.29}$$

for arbitrary $s \in (0, T]$. Invoking the Gronwall inequality once again, we conclude that $E(\xi) \equiv 0$ on $(0, T)$ and solutions of (3.2), (3.3) are unique.

Summarizing our discussions in this section, we see that we have proved the following result.

Theorem 1 *Suppose that* $\mathcal{J} \in H^1(0, T; V^*)$, $\gamma, \beta \in L^\infty(0, 1)$, $\alpha \in H^{1,\infty}(0, T; L^\infty(0, 1))$, $\kappa \in L^\infty(0, T; L^\infty(0, 1))$ *with* $\alpha, \beta, \gamma, \kappa$ *vanishing outside* $[z_1, 1]$. *Then for* $\Phi \in V = H_R^1(0, 1), \Psi \in H = L^2(0, 1)$, *we have that solutions to (3.2), (3.3) exist and are unique. These solutions satisfy* $E \in L^2(0, T; V) \cap C(0, T; H), \dot{E} \in L^2(0, T; H)$, *and* $\ddot{E} \in L^2(0, T; V^*)$. *Moreover,* $t \to E(t, 0)$ *is absolutely continuous with* $\dot{E}(\cdot, 0) \in L^2(0, T)$. *The solutions depend continuously on* $(\Phi, \Psi, \mathcal{J})$ *as maps from* $V \times H \times H^1(0, T; V^*)$ *to* $L^2(0, T; V) \times L^2(0, T; H)$.

Chapter 4

Computational Methods and Results

4.1 The Forward or Simulation Problem for the Debye Polarization Model

In this section we present computational results for both forward problem simulations and inverse problems based on the general formulation for the 1-dimensional geometry given in Section 2.3. We are concerned here with numerical results for the special case of a Debye medium Ω with $\epsilon_r(z) = \epsilon_\infty$, defining the instantaneous polarization in Ω. We first, however, formulate a Galerkin finite element approximation scheme for the system with general polarization.

We return to the differential equation (2.8) and express it in weak form by

$$\begin{aligned}\langle \mu_0 \epsilon \ddot{E}, \phi \rangle + \langle \mu_0 \sigma \dot{E}, \phi \rangle + \langle \mu_0 \ddot{P}, \phi \rangle \\ + \langle E', \phi' \rangle + \frac{1}{c} \dot{E}(t,0)\phi(0) = -\langle \mu_0 \dot{J}_s(t,\cdot), \phi \rangle\end{aligned}$$

where the polarization P is for the moment of the form given in (2.1) and the mapping to $\tilde{\Omega} = [0,1]$ has already been carried out. The term $\frac{1}{c}\dot{E}(t,0)\phi(0)$ is part of the weak form of the absorbing boundary condition.

To facilitate the computations, we scale the time variable by a factor of $c = 1/\sqrt{\epsilon_0 \mu_0}$ and polarization P by a factor of $1/\epsilon_0$ (i.e., $\tilde{t} = ct, \tilde{P} = P/\epsilon_0$). Furthermore, we assume that the permittivity of the medium Ω is a constant. The new equation in

the scaled variables (where we have dropped the overtildas on the scaled variables) is

$$\begin{aligned}&\langle \epsilon_r \ddot{E}, \phi\rangle + \eta_0\langle \sigma \dot{E}, \phi\rangle + \langle \ddot{P}, \phi\rangle \\ &\quad + \langle E', \phi'\rangle + \dot{E}(t,0)\phi(0) = -\eta_0\langle \dot{J}_s, \phi\rangle \quad \text{for } \phi \in V. \end{aligned} \tag{4.1}$$

where $\epsilon_r(z) = 1 + I_\Omega(z)(\epsilon_\infty - 1)$ is the relative electric permittivity so that $\epsilon = \epsilon_r\epsilon_0$ and the impedance of free space is defined $\eta_0 = \sqrt{\mu_0/\epsilon_0} \approx 376.73$ Ohms. Moreover, the $\langle \cdot, \cdot \rangle$ are the weighted inner products discussed in Section 2.3.

We employ a first order Galerkin finite element approximation to discretize the problem in the space variable, yielding piecewise linear approximations for $E(\cdot, z)$ and $P(\cdot, z)$. We partition the interval $[0, 1]$ uniformly at the points $\bar{z}_i^N = ih$ where $h = 1/N$ and $i = 0, \ldots, N$ and construct the standard piecewise linear spline functions $\phi_i^N(z)$ such that $\phi_i^N(\bar{z}_j^N) = \delta_{ij}$ for $i, j = 0, \ldots, N$. We omit ϕ_N^N in constructing our finite dimensional approximating subspaces $V^N = \text{span}\{\phi_0^N, \phi_1^N, \ldots, \phi_{N-1}^N\}$ so that for all the basis functions we have the essential boundary conditions $\phi_i^N(1) = 0$. The computations reported on here are also simplified by the further requirement that the material boundaries of the slab $\Omega = [z_1, 1]$ coincide with grid points. We denote the index of the left boundary z_1 of Ω by $j = L$, i.e., $\bar{z}_L^N = z_1$. Since the right edge of the material has been mapped to $z = 1$, this corresponds to the grid point $\bar{z}_N^N$.

We seek an approximate solution of (4.1) in the space $V^N \subset V = H_R^1(0, 1)$. Let E_N and P_N denote the the approximations of E and P in this space so that

$$E(t, z) \approx E_N(t, z) = \sum_{j=0}^{N-1} e_i^N(t)\phi_i^N(z) \tag{4.2}$$

$$P(t, z) \approx P_N(t, z) = \sum_{j=0}^{N-1} p_i^N(t)\phi_i^N(z). \tag{4.3}$$

By allowing both the space of solutions and space of test functions in (4.1) to be V^N in the weak form of the equation, we obtain in the usual way the Galerkin finite dimensional system of equations given by

$$(M + M_\Omega(\epsilon_\infty - 1))\ddot{e} + M_\Omega\ddot{p} + (\eta_0\sigma M_\Omega + B)\dot{e} + Ke = \eta_0\mathcal{J}. \tag{4.4}$$

for $e = (e_0^N, e_1^N, \ldots, e_{N-1}^N)$ and $p = (p_0^N, p_1^N, \ldots, p_{N-1}^N)$.

The elements of the resulting $N \times N$ finite element matrices are computed in the usual manner (for $i, j = 1, 2, \ldots, N$) by

$$\begin{aligned} M_{ij} &= \langle \phi_{i-1}, \phi_{j-1} \rangle = \int_0^1 \frac{\bar{\phi}_{i-1}\bar{\phi}_{j-1}}{1 + (\zeta - 1)I_\Omega} d\bar{z} \\ M_{\Omega ij} &= \langle \phi_{i-1}, I_\Omega \phi_{j-1} \rangle = \int_0^1 \frac{I_\Omega}{1 + (\zeta - 1)I_\Omega} \bar{\phi}_{i-1}\bar{\phi}_{j-1} d\bar{z} \\ K_{ij} &= \langle \phi'_{i-1}, \phi'_{j-1} \rangle = \int_0^1 (1 + (\zeta - 1)I_\Omega)\bar{\phi}'_{i-1}\bar{\phi}'_{j-1} d\bar{z} \\ B_{ij} &= \phi_{i-1}(0)\phi_{j-1}(0), \end{aligned} \tag{4.5}$$

while

$$\mathcal{J}_i = -\langle \dot{J}_s, \phi_{i-1} \rangle = -\int_0^1 \frac{\dot{J}_s \bar{\phi}_{i-1}}{1 + (\zeta - 1)I_\Omega} d\bar{z},$$

where the integrals are expressed in the scaled variables described in Section 2.3. We note that the variables e, p as well as the coefficient matrices should carry the index N, i.e., p^N, e^N, M^N, K^N, etc. But since this is well understood, we shall drop the notation to that given in (4.4) and (4.5) in our subsequent discussions, reminding the reader that as usual all these quantities depend on the spatial discretization index N in the obvious ways.

We have not yet imposed a particular constitutive law to govern polarization in the above formulation. We now restrict our consideration to the Debye model given in equation (2.4). Applying the same scaling in time and to P as above, (i.e., $\tilde{P} = P/\epsilon_0, \tilde{t} = ct$) we obtain the scaled Debye polarization law

$$\dot{P} + \lambda P = \epsilon_d \lambda E \quad \text{in } \Omega \tag{4.6}$$

where $\epsilon_d = \epsilon_s - \epsilon_\infty$ and $\lambda = 1/c\tau$. Since this equation only holds inside the material domain, we can equivalently multiply the entire equation by $I_\Omega(z)$ and then the Galerkin approximation results in the system of equations

$$M_\Omega \dot{p} + M_\Omega \lambda p - M_\Omega \lambda \epsilon_d e = 0. \tag{4.7}$$

The matrix M_Ω is singular (the first $L-1$ rows and the first $L-1$ columns vanish identically), so in actual computations we solve this equation for the nontrivial variables $p_i, i = L, L+1, \ldots N$ (i.e., $p_0 = p_1 = \cdots = p_{L-1} = 0$). This is equivalent to considering only the Lth through Nth elements of each vector (p, e) in the Galerkin approximation equation for P. With this tacit understanding, we may write the entire system of equations (4.4),(4.7) as

$$\begin{aligned} (M + M_\Omega(\epsilon_\infty - 1))\ddot{e} + M_\Omega\ddot{p} & \\ + (\eta_0\sigma M_\Omega + B)\dot{e} + Ke &= \eta_0\mathcal{J} \qquad (4.8) \\ \dot{p} + \lambda p - \lambda\epsilon_d e &= 0. \qquad (4.9) \end{aligned}$$

By substituting equation (4.9) and its derivative into equation (4.8) we obtain an equivalent system of equations

$$\begin{aligned} (M + (\epsilon_\infty - 1)M_\Omega)\,\ddot{e} + (\lambda\epsilon_d M_\Omega + \eta_0\sigma M_\Omega + B)\dot{e} & \\ + (-\lambda^2\epsilon_d M_\Omega + K)e + \lambda^2 M_\Omega p &= \eta_0\mathcal{J} \qquad (4.10) \\ \dot{p} + \lambda p - \lambda\epsilon_d e &= 0. \end{aligned}$$

This can be written as a first order system in the composite variable $x = (e, p, \dot{e})$ as

$$\bar{M}\dot{x} + \bar{K}x = F$$

or,

$$\begin{bmatrix} I & & \\ & I & \\ & & M_1 \end{bmatrix} \begin{bmatrix} \dot{e} \\ \dot{p} \\ \dot{e} \end{bmatrix} + \begin{bmatrix} 0 & 0 & -I \\ -\lambda\epsilon_d I_{LR} & \lambda I_{LR} & 0 \\ M_2 & M_3 & M_4 \end{bmatrix} \begin{bmatrix} e \\ p \\ \dot{e} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \eta_0\mathcal{J} \end{bmatrix} \qquad (4.11)$$

where

$$\begin{aligned} M_1 &= M + (\epsilon_\infty - 1)M_\Omega \\ M_2 &= -\lambda^2\epsilon_d M_\Omega + K \\ M_3 &= \lambda^2 M_\Omega \\ M_4 &= M_\Omega(\lambda\epsilon_d + \eta_0\sigma) + B \end{aligned}$$

and I_{LR} is the $N \times N$ identity matrix where the ones have been replaced with zeros in rows 1 through $L-1$.

We compute an approximate solution to this differential system with the standard Crank-Nicholson scheme, which is a member of a single parameter (θ) family of schemes. Briefly, this can be summarized as follows. For a given value of θ and a step size k, the family of schemes applied to the differential equation: $\dot{x} = f(t,x)$ yields the sequence of iterates $x_n \approx x(t_n) = x(nk)$ where

$$x_{n+1} = x_n + k f_{n+\theta} \tag{4.12}$$

and

$$f_{n+\theta} = (1-\theta) f(t_n, x_n) + \theta f(t_{n+1}, x_{n+1}).$$

This family includes the Euler scheme when $\theta = 0$, the Crank-Nicholson scheme when $\theta = 1/2$ and the implicit Euler when $\theta = 1$. Since x_{n+1} appears on both sides of equation (4.12), the method is implicit unless $\theta = 0$. Since our system is linear, it can be solved directly for the value of x_{n+1} even in the case $\theta \neq 0$. Applying this to our matrix system, we obtain another matrix problem for the iterations

$$x_{n+1} = x_n + k y_n$$

where

$$(\bar{M} + k\theta\bar{K}) y_n = -\bar{K} x_n + F_{n+\theta} \tag{4.13}$$

and $x_0 = 0$, since the material is assumed to be initially electrically inactive. Equation (4.13) is reduced through block-Gaussian elimination to a block upper-triangular system of size $3N - L + 1$, in which only a single block of size N needs to be factored. The LU factorization of this block can be computed once and used throughout the computation. Computational experiments with different values of θ did not produce any substantial improvement in accuracy over $\theta = 1/2$ for our particular systems. The Crank-Nicholson scheme is also known to be unconditionally stable and second order accurate. Unlike the explicit ($\theta = 0$) and ordinary implicit ($\theta = 1$) members of

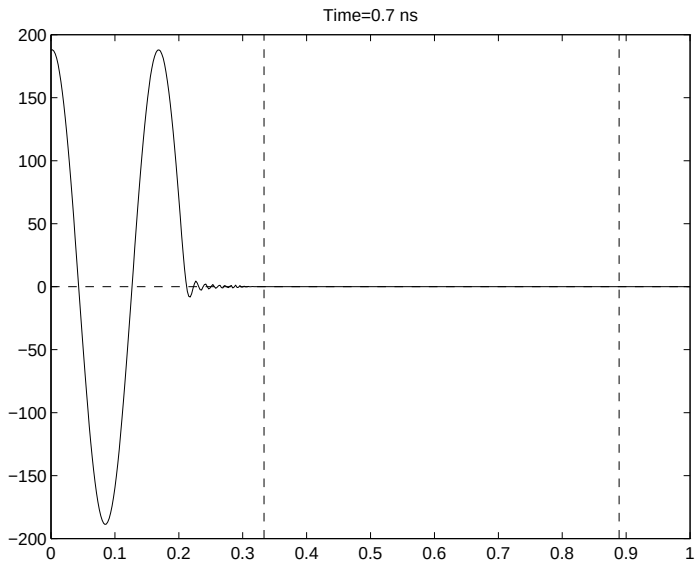

Figure 4.1: Debye model simulation ($t = 0.7$ ns)

this family, Crank-Nicholson is also not excessively dissipative (see [Glo, p100-101]). Hence in the numerical results described below all calculations were carried out with the standard Crank-Nicholson time stepping scheme.

We report graphically a sample of results from our forward simulations for the model with Debye polarization. Figures 4.1 through 4.4 depict (through time snapshots at $t = 0.7, 5.0, 7.0, 10.0$ ns) the propagation of an electromagnetic wave through a material slab lying in $z \in (.33, .89)$. The material parameters are: $\sigma = 1.0 \times 10^{-2}$ Ohm^{-1}, $\tau = 1.0 \times 10^{-11}$ seconds, $\epsilon_s = 35, \epsilon_\infty = 5, \omega = 2\pi \times 1.8$ GHz. The numerical method is as described above, with the results depicted in the unscaled (i.e., in the original scales) spatial (z) and time (t) axes.

In Figure 4.1 the incoming wave generated by the current source at $z = 0$ has yet to reach the left edge of the material at $z_1 = 1/3$. In Figure 4.2 the signal has subsequently been partially reflected and partially transmitted. The reflected part of the field is the first part to be measured in the inverse problems discussed in the next

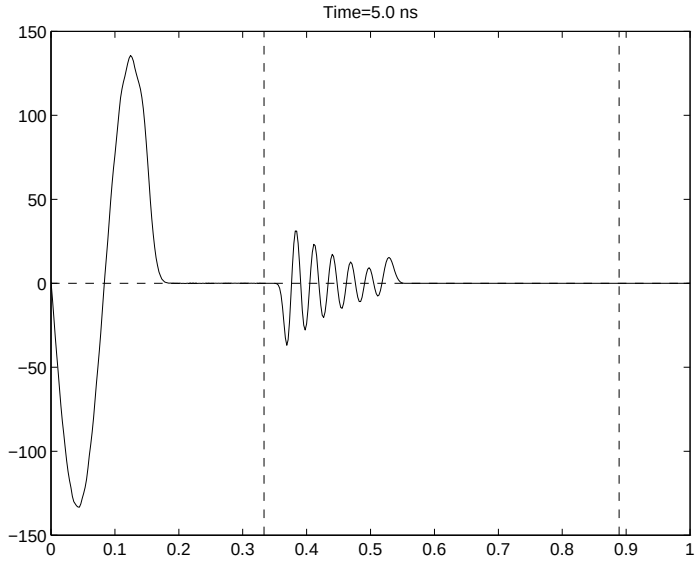

Figure 4.2: Debye model simulation ($t = 5.0$ ns)

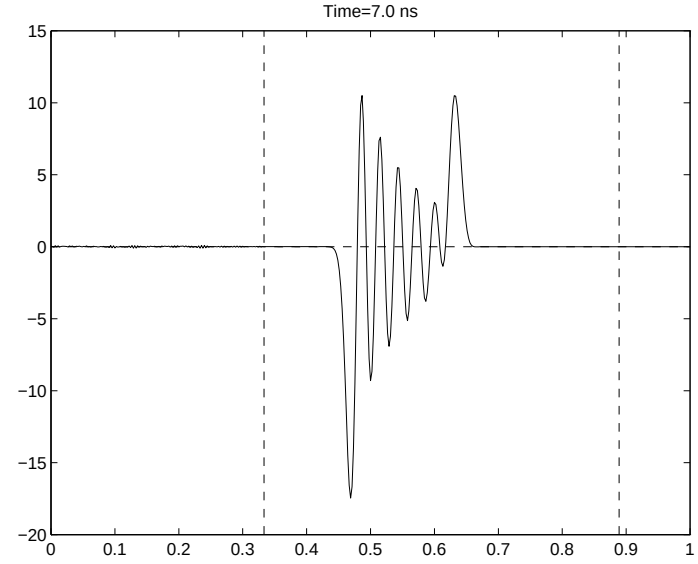

Figure 4.3: Debye model simulation ($t = 7.0$ ns)

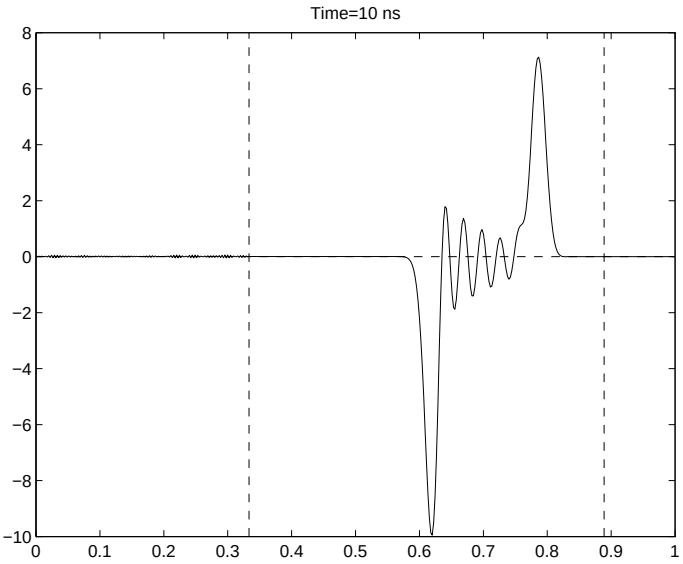

Figure 4.4: Debye model simulation ($t = 10.0$ ns)

section. In the transmitted part we see the formation of the signal precursor (the Brillouin precursors— see [APM, Bri]), which becomes more pronounced in Figures 4.3 and 4.4. These simulations were performed on an interval of length 1 with $N = 450$ and the time step was $k = 1 \times 10^{-4}$. Comparison of these (and other simulations) with independently generated solutions (finite difference and Fourier series) demonstrated the general accuracy and efficiency of the piecewise linear/Crank-Nicholson approximation methods in forward simulations and we turned next to the use of these ideas in inverse problem techniques.

4.2 The Inverse or Estimation Problem with the Debye Model

The objective of the inverse problem is the reconstruction of the values of the parameters in the polarization model and the boundary geometry using information obtained

through a scattering experiment of the type described in Section 2.1. Observations from the experiment are limited to sampled values of the electric field at selected points outside the material domain Ω. The estimation problem is to minimize a suitable measure of the difference between the simulated prediction and a set of data taken from experiments. The goal of our investigation here is the test the feasibility of this approach for the identification of dielectric and geometric parameters.

We use the same physical problem as in the forward problem described in Section 2.3; this involves a homogeneous slab obeying the Debye model of polarization and a planar electromagnetic interrogating signal. Therefore our observations and data consist of scalar values representing the $\hat{\imath}$ component of the electric field. The experimental observations, consist of the value of the electric field at $z = 0$ at uniform intervals in time $\bar{t}_i = \Delta T i$. Let E_i denote the data we seek to reconstruct and $E(t, z; \vec{q})$ be the electric field arising from a scattering experiment with dielectric and geometric parameters $\vec{q}$. The inverse problem is performed by minimizing the l^2 difference between the data and the simulation results.

$$\min_{\vec{q} \in Q} J(\vec{q}) \quad \text{where} \quad J(\vec{q}) = \sum_{i=1}^{S} |E(\bar{t}_i, 0; \vec{q}) - E_i|^2$$

where S is the number of sample data points. The set of admissible parameters Q is chosen to enforce limitations on the parameters that arise from physical or geometric considerations. The formulation of physical problems generally requires that physical parameters be non-negative. Geometric bounds will be determined according to how the boundary is represented by the parameters, and will reflect both mathematical constraints and physical limitations on the boundary. This set can be made compact by providing both upper and lower bounds for each member.

To test the feasibility of the estimation approach we produce synthetic data for the observations E_i by adding random noise to the results of the simulation with a known set of parameters. The absolute magnitude of the noise is relative to the size of the signal, reflecting the relative nature of uncertainties in measurements. Letting E_i be the data sampled from the solution with the true parameters, i.e., $E_i = E(\bar{t}_i, 0; \vec{q}^*)$,

we define $\bar{E}_i = E_i(1+\chi\eta_i)$ where the η_i are independent normally distributed random variables with mean zero and variance one. The coefficient χ is chosen to adjust the relative magnitude of the noise. We express the magnitude of the noise as a percentage of the size of the signal by taking two times the variance as the size of the random variable. Hence $\chi = 0.05$ corresponds to 10% noise and $\chi = 0.025$ to 5% noise.

The feasibility of the inverse problem is measured by how successful it is at recovering the original values $\vec{q}^*$ and the sensitivity of the results with respect to the magnitude of the noise χ. In the absence of noise, an exact match for the parameter values has an error of zero (or roughly machine precision, allowing for non-essential differences in the method of computation). Minimizing $J(\vec{q})$ is performed though an l_∞ trust region adaptation of Newton's method, using a BFGS secant update for the approximation of the Hessian of the objective function $\frac{\partial^2}{\partial q_i \partial q_j} J(\vec{q})$.

The parameters arising in the Debye model of polarization are the conductivity σ, the infinite and static limits of the dielectric permittivity $\epsilon_\infty, \epsilon_s$ and the relaxation time τ. (Strictly speaking, the conductivity σ is not part of the polarization model, but for convenience we will always group it with the other parameters which describe the dielectric properties of the material.) We will directly identify the related set of parameters $\vec{q} = (\sigma, \epsilon_s, \epsilon_d, \lambda)$ where $\epsilon_d = \epsilon_s - \epsilon_\infty$ and $\lambda = 1/c\tau$. These parameters appear as coefficients in the scaled version of the equations and early simulation results indicated that the optimization problem is better posed when expressed in terms of λ rather than τ. This also resolves problems arising from the difference in scale between τ and the other parameters, since the typical magnitude of λ is around 400. In a subsequent problem we will add the identification of the thickness of the slab $d = z_2 - z_1$. In the scaled version of the equations this is done through identifying the scaling parameter $\zeta = (1 - z_1)/(z_2 - z_1)$ which appears in the definition of the weighted inner products.

Sample Results

We present results from the inverse problem of identifying the dielectric parameters in the Debye model. The data to be identified are generated with the Debye model using the following ("true") parameter values:

$$\begin{aligned} \sigma &= 1 \times 10^{-5} \\ \tau &= 8.1 \times 10^{-12} \\ \epsilon_s &= 80.1 \\ \epsilon_\infty &= 5.5 \end{aligned}$$

These parameters are considered reasonable for modeling the polarization behavior of water. The carrier frequency of the interrogating signal is 1.2GHz (hence the angular frequency is $\omega = 2\pi \times 1.2 \times 10^9 = 7.54 \times 10^9$ rad/s) and the duration of the window is 1.67 ns, which is two complete periods. The data consist of 100 measurements of the electric field taken at $z = 0$ every 0.06 ns from $t = 0.06$ ns to $t = 6$ ns. The width of the slab is a fixed quantity sufficient to ensure that the portion of the interrogating signal transmitted through the surface at $z = z_1$ has not returned to $z = 0$ after a subsequent reflection off the back boundary.

Eight different attempts are made at the inverse problem, each with a different level of random noise added to the data. The initial values of parameters used in all of the inverse problems are $\sigma_0 = 1.5 \times 10^{-5}, \tau_0 = 10.0 \times 10^{-12}, \epsilon_{s0} = 73.1, \epsilon_{\infty 0} = 6.0$. The results are summarized in Table 4.1.

We note that only the parameter ϵ_s is consistently recovered in this inverse problem. While some of the values of τ are close, the converged values of σ and ϵ_∞ are substantially off at all levels of noise. A partial explanation for the differing sensitivities can be found by examining the equations which give rise to the wave equation and polarization equation. Consider the polarization equation,

$$\tau \dot{P} + P = \epsilon_0(\epsilon_s - \epsilon_\infty)E. \tag{4.14}$$

The small magnitude of τ suggests that $P \approx \epsilon_0(\epsilon_s - \epsilon_\infty)E$. We use this as the basis

Test	% Noise	σ	τ	ϵ_s	ϵ_∞	residual
True values:		1.0×10^{-5}	8.1×10^{-12}	80.1	5.5	
Initial values:		1.5×10^{-5}	10.0×10^{-12}	73.1	6.0	
1	0.0	9.96×10^{-6}	8.10×10^{-12}	80.10	5.51	6.50×10^{-10}
2	0.5	0	1.23×10^{-11}	80.20	29.74	2.18×10^{-2}
3	1.0	0	9.18×10^{-12}	80.15	13.89	0.128
4	1.5	1.14×10^{-2}	2.89×10^{-11}	81.18	60.02	0.263
5	2.0	0	6.49×10^{-12}	80.46	1.00	0.474
6	2.5	7.34×10^{-3}	1.93×10^{-11}	80.04	48.68	0.575
7	3.0	0	1.30×10^{-11}	79.95	31.10	0.835
8	5.0	5.56×10^{-2}	3.77×10^{-12}	78.82	28.07	2.035

Table 4.1: Estimated Parameters in Debye Inverse Problem. Test 1

of an approximation to P. Let $P_s = \epsilon_0(\epsilon_s - \epsilon_\infty)E$ and $P = P_s + P_\tau$. The remaining polarization P_τ satisfies the new differential equation

$$\tau \dot{P}_\tau + P_\tau = -\tau\epsilon_0(\epsilon_s - \epsilon_\infty)\dot{E}. \tag{4.15}$$

Note that this is the same equation as for the original polarization with a different driving function. We substitute into the equation for the electric flux density and find

$$\begin{aligned} D &= \epsilon_0\epsilon_\infty E + P \\ &= \epsilon_0\epsilon_s E + P_\tau \end{aligned}$$

When the wave equation is derived from this new expression for D, ϵ_s replaces ϵ_∞ as the coefficient of the $\ddot{E}$ term and the parameter ϵ_∞ only appears in the differential equation for P_τ. It remains to argue that the magnitude of this polarization term is much smaller than that of the original P. We make an argument for this based on the roughly periodic nature of the electric field. At any point, when the electric field is not zero (i.e., outside the windowed pulse) the electric field is approximately periodic with a known angular frequency ω. From this we conclude, very approximately, that $|\dot{E}| \approx |\omega E|$. Comparing the magnitudes of the driving terms in the original (for P)

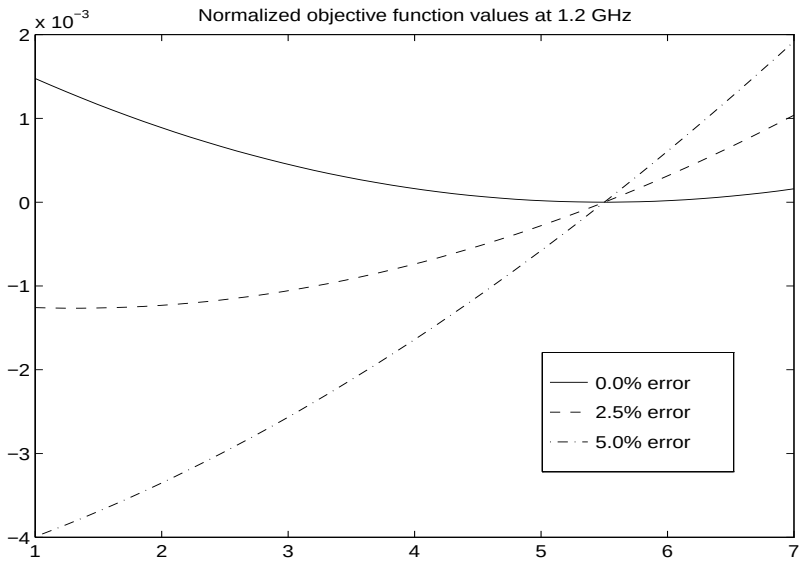

Figure 4.5: Plot of $J(\vec{q}) - J(\vec{q}^*)$ versus ϵ_∞

and new (for P_τ) polarization equations we have $|E|/|\tau\dot{E}| \approx 1/\omega\tau$. For the parameter values $\tau = 8.1\times10^{-12}$ s and $\omega = 7.54\times10^9$ rad/s we would expect that that magnitude of P is about 16 times as great as P_τ. Furthermore, ϵ_∞ is many times smaller than ϵ_s and therefore affects the coefficient of the driving term less. This suggests that the effect on the solutions to the forward problem of the parameter ϵ_∞ is dwarfed by that of ϵ_s.

We illustrate this idea graphically by plotting the objective function $J(\vec{q})$ as a function of ϵ_∞ alone. The plots of the objective functions in Figures 4.5 and 4.6 are normalized by subtracting the value of $J(\vec{q}^*)$. When no noise in present in the data, this value is zero. The values when noise is present are given in Table 4.2.

Comparing the two figures, we can see that the objective function is more sensitive to ϵ_∞ for the larger frequency and that the minimizing value of ϵ_∞ is closer to the true value. For the case of 5% noise and $\omega = 2\pi \times 1.2$ rad/s the minimizing value of ϵ_∞ has apparently slipped below the minimum value of $\epsilon_\infty = 1$, the lowest value

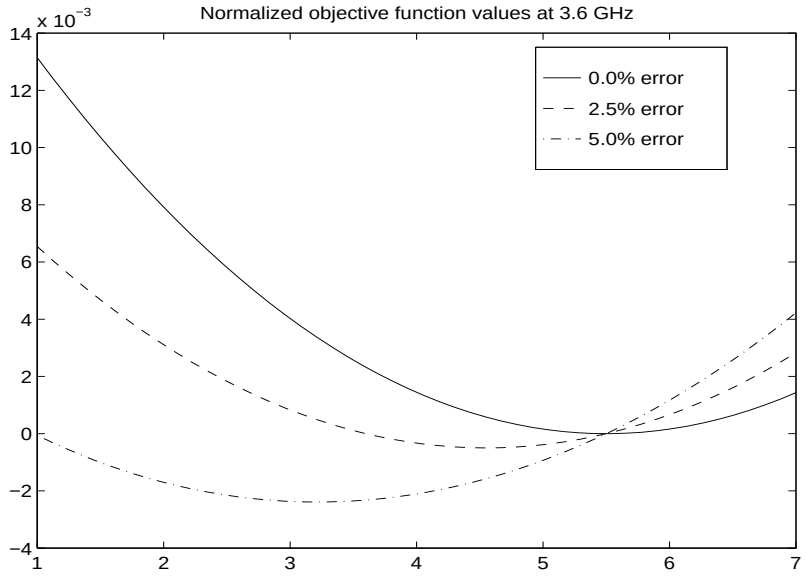

Figure 4.6: Plot of $J(\vec{q}) - J(\vec{q}^*)$ versus ϵ_∞

	frequency (GHz)	
Noise level	1.2	3.6
2.5%	0.772	5.30
5.0%	3.01	21.20

Table 4.2: $J(\vec{q}^*)$ in the presence of noise

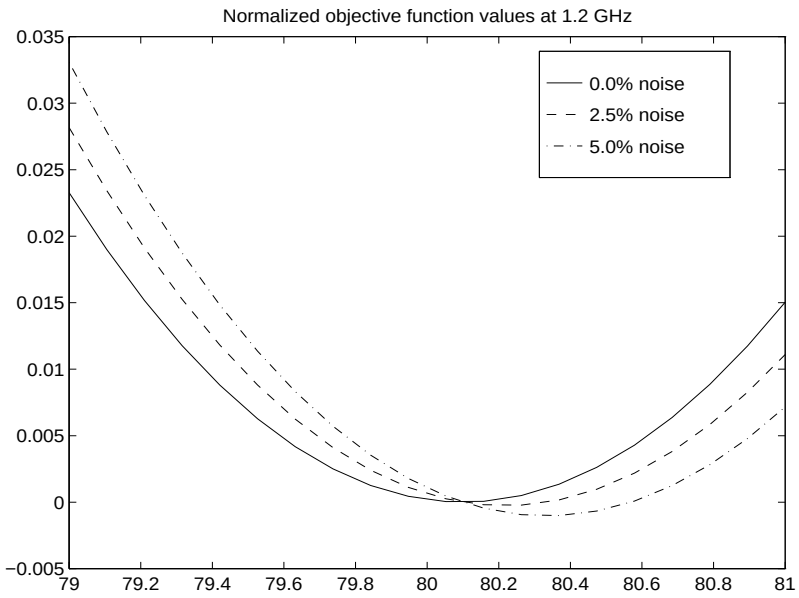

Figure 4.7: Plot of $J(\vec{q}) - J(\vec{q}^*)$ versus ϵ_s

of ϵ_∞ allowable on physical grounds. For comparison, we also show the dependence of the objective function on the parameters ϵ_s in Figure 4.7 and σ in Figure 4.8. Even in a comparatively small range of values for ϵ_s, we see that the dependence is much stronger and that the minimum value is not perturbed as far as ϵ_∞. On the other hand, the dependence of the objective function on σ is moderate for a range of values which is several orders of magnitude larger than the value of σ itself. Hence, we expect that σ is a difficult parameter to recover accurately, when accuracy is measured relative to the size of the parameter.

The new polarization equation (4.15) also suggests that it may be difficult to separate the parameters τ from ϵ_∞, since the same analysis as before leads us to conclude that $P_\tau \approx -\tau\epsilon_0(\epsilon_s - \epsilon_\infty)E$. Here, τ and ϵ_∞ appear together in the coefficient of the driving term and any other influence of τ on the solution has been ruled out by this approximation. While it should be possible to estimate the size of the driving term's coefficient, it would not be possible to identify particular values for ϵ_∞ and

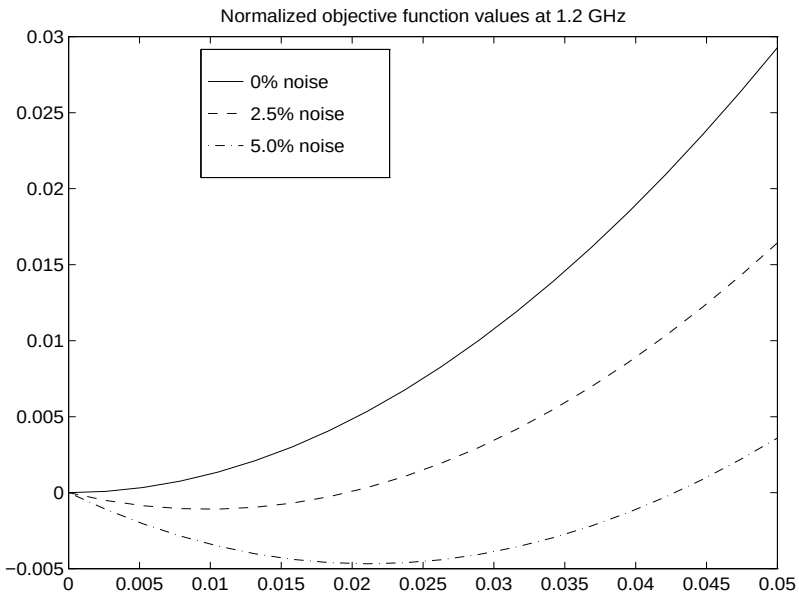

Figure 4.8: Plot of $J(\vec{q}) - J(\vec{q}^*)$ versus σ

fixed ϵ_∞	estimated τ	estimated ϵ_s	residual	$\tau(\epsilon_s - \epsilon_\infty)$
1.0	7.62×10^{-12}	79.96	3.94×10^{-4}	6.02×10^{-10}
5.5	8.10×10^{-12}	80.10	6.36×10^{-11}	6.04×10^{-10}
10.0	8.64×10^{-12}	80.26	1.61×10^{-4}	6.07×10^{-10}

Table 4.3: Parameters resulting from incorrect ϵ_∞

τ. (The parameter ϵ_s, which is also present in the coefficient, is distinguished by its appearance in the wave equation.) This observation is borne out by results from another inverse problem in which the values σ and ϵ_∞ are held fixed. The conductivity σ is fixed at zero and the interrogating frequency is 3.6 GHz. Performing the zero-noise inverse problem, we obtain different values of τ according to the fixed value of ϵ_∞. These results are summarized in Table 4.3.

The residual column of Table 4.3 indicates that it is possible to get an acceptably low residual value with parameters far from the "true" values which generated

Test	% Noise	σ (fixed)	τ	ϵ_s	ϵ_∞ (fixed)	residual
True values:		1.0×10^{-5}	8.1×10^{-12}	80.1	5.5	
Initial values:		1.5×10^{-5}	10.0×10^{-12}	73.1	6.0	
1	0.0	0	7.62×10^{-12}	79.96	1.00	3.942×10^{-3}
2	0.5	0	7.64×10^{-12}	80.37	1.00	0.208
3	1.0	0	7.65×10^{-12}	80.79	1.00	0.834
4	1.5	0	7.66×10^{-12}	81.21	1.00	1.877
5	2.0	0	7.67×10^{-12}	81.63	1.00	3.338
6	2.5	0	7.68×10^{-12}	82.06	1.00	5.217
7	3.0	0	7.70×10^{-12}	82.49	1.00	7.513
8	5.0	0	7.74×10^{-12}	84.24	1.00	20.87

Table 4.4: Estimated Parameters in Debye Inverse Problem. Test 2

the data. The last column shows that the coefficient of the driving term in the polarization equation (4.15) is being accurately reconstructed each time, even if the particular parameter values are wrong. This suggests that the difficulty surrounding the identification of the parameters ϵ_∞ and τ arises because these two parameters are strongly coupled together in their influence on the objective function. The relatively low sensitivity of the objective function to these parameters (compared to ϵ_s) further suggests that parameter values obtained by arbitrarily fixing one of the parameters and identifying a corresponding value for the other will yield parameters which capture the dynamics of the problem accurately (i.e., yield a small residual). To test this approach we perform the optimization problem with the parameter ϵ_∞ fixed at the value 1.0 and the conductivity σ fixed at zero. The results are summarized in Table 4.4.

In the event that the exact value of ϵ_∞ is known, we can fix the parameter at this value in the optimizations. The results of performing the inverse problem with ϵ_∞ fixed at its correct value of 5.5 are shown in Table 4.5. We notice that except in the case of zero noise that the residuals are slightly *higher* when the correct value of ϵ_∞ is used than when $\epsilon_\infty = 1.0$ is used. This is explained by our observation in Figures 4.5 and 4.6 that the presence of noise uniformly lowers the optimal value of ϵ_∞.

Test	% Noise	σ (fixed)	τ	ϵ_s	ϵ_∞ (fixed)	residual
True values:		1.0×10^{-5}	8.1×10^{-12}	80.1	5.5	
Initial values:		1.5×10^{-5}	10.0×10^{-12}	73.1	6.0	
1	0.0	0.00	8.11×10^{-12}	80.5	5.5	6.364×10^{-10}
2	0.5	0.00	8.12×10^{-12}	80.93	5.5	0.209
3	1.0	0.00	8.13×10^{-12}	81.35	5.5	0.835
4	1.5	0.00	8.14×10^{-12}	81.77	5.5	1.879
5	2.0	0.00	8.14×10^{-12}	82.20	5.5	3.341
6	2.5	0.00	7.68×10^{-12}	82.06	5.5	5.220
7	3.0	0.00	7.70×10^{-12}	82.49	5.5	7.517
8	5.0	0.00	7.74×10^{-12}	84.24	5.5	20.88

Table 4.5: Estimated Parameters in Debye Inverse Problem. Test 2

Identification of Material Depth

Identifying the depth of the material with this method proved to be more difficult than identifying other parameters. We add the scaling parameter to the list of parameters to be identified, so $\vec{q} = (\sigma, \epsilon_d, \epsilon_\infty, \lambda, \zeta)$ and consider a slightly different inverse problem. In this case, the material is sufficiently thin and the duration of the data collection sufficiently long so that the part of the interrogating signal transmitted through the medium had returned after being reflected off the metallic back boundary. This is clearly necessary in any attempt to recover the depth of the sample.

We see in in Figure 4.9 the complicated dependence of $J(q, d)$ on d, owing to the oscillatory nature of the time domain data. (The results are presented in terms of the actual thickness d instead of the scaling parameter ζ). Here, $J(\vec{q}^*, d)$ is plotted for various values of d with the other parameters held fixed at their true values. The true depth of the material is $d = 0.05$, corresponding to the global minimum of $J(\vec{q})$. (This example uses the parameter values for water given above.)

Figure 4.10 depicts one cause of the multiple local minima apparent in Figure 4.9. With the incorrect value of d, the return of the transmitted and subsequently reflected part of the interrogating signal is delayed by two full periods of the input signal. This

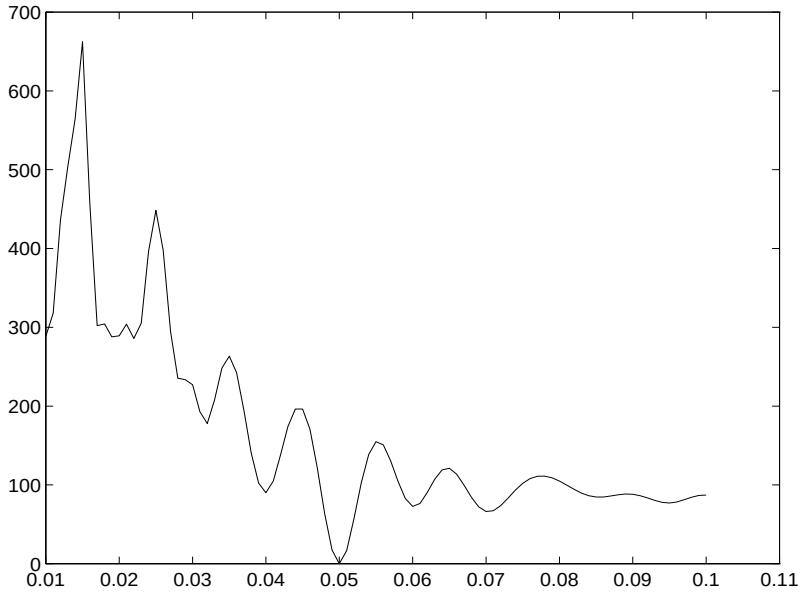

Figure 4.9: Error as a function of material depth.

brings it back into partial synchronization with the data, giving it a smaller error than nearby values where the data is out of phase. This produces a local minima as seen in Figure 4.9.

Optimization results using the function $J(\vec{q})$ where $\vec{q}$ includes the scaling parameter ζ were very poor, even when only attempting to identify the depth d. Solutions tended to incorrect local minima for all but a very narrow range of initial values of d. This led us to develop another procedure for identifying depth, based on identifying the return time of the transmitted part of the interrogation signal.

We separate the geometric parameter ζ from the rest of the parameters in $\vec{q}$ to clarify parts of the following discussion. Here, $\hat{q}$ contains just the physical (material) parameters and $\vec{q} = (\hat{q}, \zeta)$. We will interchangeably discuss estimation of the parameters ζ and d with the understanding that the actual optimization is always performed over ζ and that the two parameters are related through $\zeta = (1 - z_1)/d$.

Let $t_r(\hat{q}, d)$ be the time at which the signal which penetrates Ω first returns to

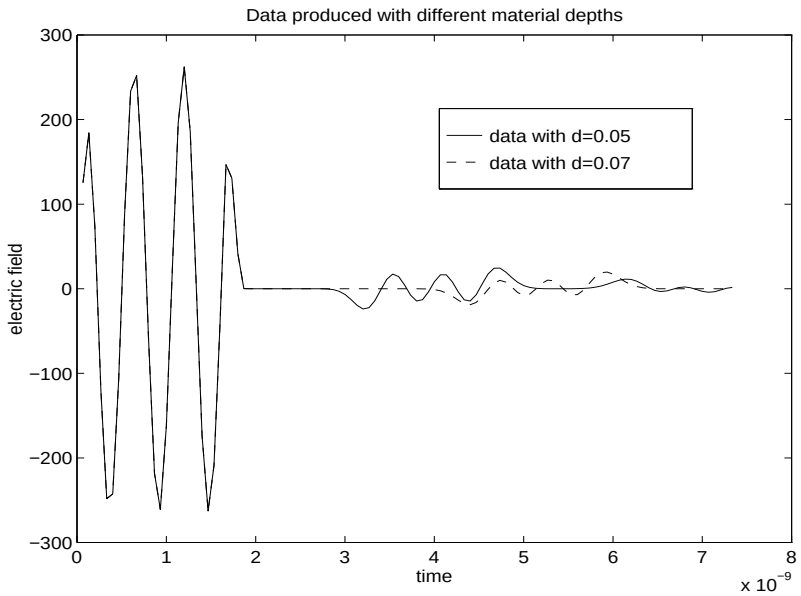

Figure 4.10: Illustration of local minima at $d = 0.07$

$z = 0$ after being transmitted through the medium and reflected off its back surface, expressed as a function of the material parameters and the depth of the sample. Let Δ and δ be positive constants with values to be determined and t_f^s be the duration of the interrogation signal. We approximate the return time by: $t_r(q,d) \approx \tilde{t}_r(q,d) = \bar{t}_{\bar{\imath}}$ where $\bar{\imath}$ is the first index such that:

$$E_i - E_{i-1} \geq \Delta(\bar{t}_i - \bar{t}_{i-1}) \tag{4.16}$$

$$\bar{t}_i \geq (1+\delta)(2z_1 + t_f^s). \tag{4.17}$$

Equation (4.16) is satisfied when the finite difference approximation to $E_t(t,0)$ is greater than the constant Δ. The constraint in equation (4.17) prevents the test from detecting the first reflection off the surface of Ω, which is over at $t = 2z_1 + t_f^s$. (Note that the time is expressed in the scaled variable.) The parameter δ is chosen to delay detection further in order to avoid detecting numerical noise or the decaying electric field which trails behind the reflection when the value of the conductivity σ

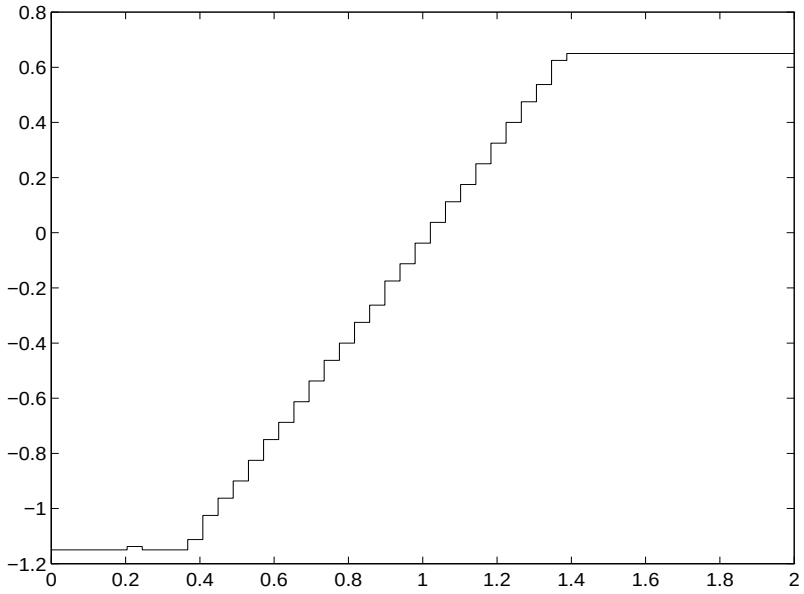

Figure 4.11: $f_t(d)$ for example problem.

is large. Typical values of these parameters which give good results are $\Delta = 20.0$ and $\delta = 0.05$.

We use the estimated return time $\tilde{t}(\hat{q}, d)$ to identify the true material depth d^*. Let $\tilde{t}(\hat{q}^*, d^*)$ be the estimated return time (satisfying equations (4.16) and (4.17)) for the true data E_i, generated with the true parameter values $\hat{q}^*$ and material depth d^*. We then look for a root of the equation $f_t(d) = \tilde{t}(q^*, d) - \tilde{t}(q^*, d^*)$, which represents the difference in estimated return times between the simulation and the data.

The function $f_t(d)$ corresponding to the example given in detail below is plotted in Figure 4.11. The plateau of values for large d occurs because the deep reflection no longer returns before the end of the simulation and the detection algorithm returns t_f, yielding $f_t(d) = t_f - \tilde{t}_r(q^*, d^*)$

Because $f_t(d)$ is a discontinuous function of a single variable, we employ Brent's method [PTVF, p.352] to approximate the root, since it does not rely on derivative

Noise Level %	Converged to
0.5	0.099996
10.0	0.099926
20.0	0.09985
30.0	0.09978
40.0	0.09972
100.0	0.09934

Table 4.6: Results of depth estimation

information. The piecewise constant character of $f_t(d)$ also limits the accuracy available in the estimate of d. Therefore, we adopt a two-step approach to finding the final value of this parameter.

1. Use Brent's method to find an estimate d_1 of the root of $f_t(d)$.

2. Use $\zeta_1 = (1-z_1)/d_1$ as an initial value for the continuous optimization of $J(q^*, d)$ over the single parameter ζ.

Results The two step approach described above is extremely effective. For a test problem using the physical parameters for water ($\sigma = 1.0 \times 10^{-5}, \epsilon_s = 80.1, \epsilon_\infty = 5.5, \tau = 8.1 \times 10^{-12}$) and $\omega = 2\pi \times 2.33$ GHz, $t_f^s = 1.66$ ns, $t_f = 8.33$ ns and $d^* = 0.1$ a wide variety of initial values of d in the range $(0, 1.0)$ lead to convergence to the correct value. In these simulations, $\Delta T = 4.66 \times 10^{-3}$ ns and the parameters in the signal return detection are $\Delta = 20.0, \delta = 0.05$.

The identification of the material depth is also very insensitive to noise in the data. We tested the sensitivity of the problem by adding normally distributed random noise of various magnitudes to the data set as in Section 4.2 and repeated the inverse problems using the initial value $d_0 = 0.02$, with the results obtained shown in Table 4.6.

Here we can see that noise as large as the amplitude of the original signal does not affect the depth estimation very much. This robustness of the depth estimate is

due in part to the nature of the noise. Since it is relative noise, it has little effect on the data points between the surface reflection and the return of the signal which penetrated the material since these values are all nearly zero. Thus the return time of the signal is not effected, since the point at which the data has its first substantial jump is the same. Furthermore, the minimum of $J(\vec{q})$ is still attained by the value of d for which the penetrating part of the signal returns at the correct time, causing the zero and non-zero parts of the data to match.

Estimating Depth and Physical Parameters

The method described above for estimating the depth d is limited by the need to know the exact values of the physical parameters $\hat{q}^*$ to form the function $f_t(d) = \tilde{t}(\hat{q}^*, d) - \tilde{t}(\hat{q}^*, d^*)$. We eliminate this requirement by modifying the algorithm and obtaining the following three step process.

1. Estimate the physical parameters by minimizing $J(\vec{q})$ over the physical parameters $\hat{q}$ only. Do this over a data set sufficiently short so that it only contains the exterior surface (i.e., $z = z_1$) reflection. Choose $\zeta_0 = (1 - z_1)/d_0$ such that the penetrating signal does not return from the back boundary in any of the simulations. To do this, choose $t'_f < t_r(\hat{q}^*, d^*)$ and $t'_f < t_r(\hat{q}, d_0)$ for a wide range of values of $\hat{q}$ and d_0. Estimate $\hat{q}_1$ only using the data E_i where $\bar{t}_i < t'_f$.

2. Using the estimate $\hat{q}_1$ from step 1, find the root of $f_t^1(d) = \tilde{t}(\hat{q}_1, d) - \tilde{t}(\hat{q}^*, d^*)$. Call this estimate d_1

3. Use the estimated value $\zeta_1 = (1 - z_1)/d_1$ from step 2 and the estimate $\hat{q}_1$ from step one as an initial estimate for minimizing $J(\vec{q})$.

To accomplish step 1, we use the physical property that the speed of propagation of waves in the media cannot be any faster than is it in vacuum. Since the deep reflection must transverse the distance z_1 twice at speed one and the depth of the material twice at no more than speed one, $t_r(\hat{q}, d) > 2(z_1 + d)$ for all parameter values $\hat{q}$.

We need to choose the reduced test time t'_f so that it is after the end of the first reflection and before the beginning of the deep reflection. Hence,

$$2z_1 + t_f^s \leq t'_f \leq 2z_1 + 2d \tag{4.18}$$

Using this conservative estimate, we can guarantee the existence of a suitable test time t'_f where $d > t_f^s/2$ and where $t_f^s \in [2z_1 + t_s^f, 2z_1 + 2d]$. In practice, we can relax the constraint on d because the typical speed on propagation in these materials is much less than 1. This is the case in the example given in the next section.

Results

Our example for this inverse problem uses the following physical parameters for the generation of data: $\sigma^* = 1.5 \times 10^{-5}$, $\tau^* = 8.1 \times 10^{-12}$, $\epsilon_s^* = 80.1$, $\epsilon_\infty^* = 5.5$, $d^* = 0.05$. These parameters will be estimated in the inverse problem. The leading edge of the material is at $z_1 = 0.25$ and the simulation is run until $t_f = 7.33$ ns, which is sufficient time for the transmitted part of the signal to return to $z = 0$ after subsequent reflections off the far boundary. The duration of the input signal is $t_f^s = 1.66$ ns and its frequency is $\omega = 2\pi \times 1.8$ GHz, yielding three full oscillations. The data is sampled 110 times over the duration of the simulation, hence $\Delta T = 0.0667$ ns. The resulting data collected at $z = 0$ is plotted in Figure 4.12. The transmitted pulse can be seen passing through $z = 0$ over the time span 4.3 ns to 6.7 ns.

Notice that the right inequality in equation (4.18) is not satisfied by this value of d. We can see from Figure 4.12 that the slow propagation of the signal through the medium nevertheless results in a gap between the end of the first reflection and the beginning of the transmitted signal.

The first stage of the optimization is carried out over half of the data up to the time $t'_f = 1.1$. The data was generated numerically using 600 basis elements in the spatial (finite element) approximation. As before, random noise is added to the results of the forward problem to generate synthetic data for the inverse problem. Various initial values are used to test the range of convergence for the optimization problem.

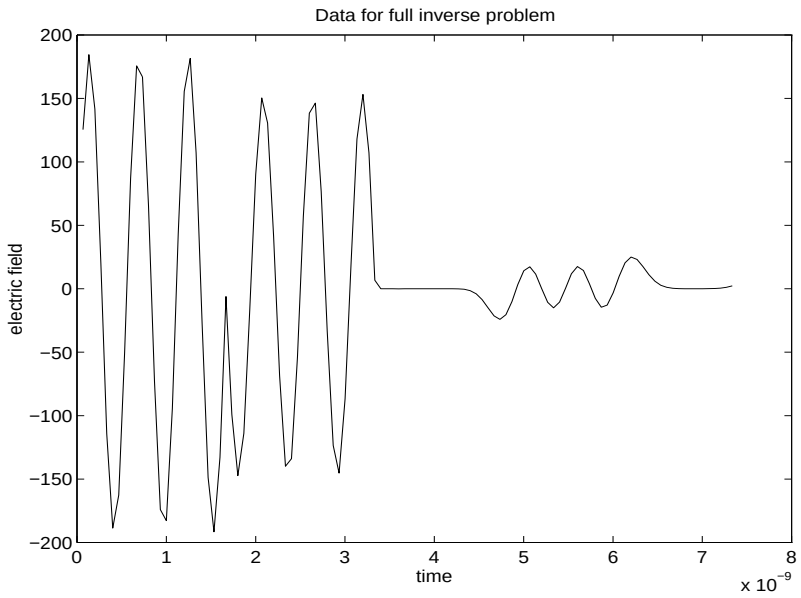

Figure 4.12: E_i for inverse problem.

The results of one such test are given in Table 4.7.

As before, we see that the estimates of σ and ϵ_∞ are not very good, while ϵ_s and τ are comparatively robust with respect to noise in the data. Further, the estimated depth of the sample is largely insensitive to noise. Performing the experiment again with a different initial value yields similar results as given in Table 4.8.

Aside from the unreasonable values (underlined in Table 4.8) which appear in the case of 2.0% noise, these results resemble those in the previous trial, especially in the values of ϵ_s and τ. As before, the depth measurement is quite accurate. The results of the 2.0% trial indicate that the method as presented here is not perfectly robust, and some caution interpreting results is called for.

% Noise	σ	τ	ϵ_s	ϵ_∞	d	residual
True values:	1.0×10^{-5}	8.1×10^{-12}	80.1	5.5	0.05	
Initial values:	1.0×10^{-5}	8.2×10^{-12}	81.0	5.3	0.055	
0.0	2.0×10^{-5}	8.07×10^{-12}	80.09	5.19	0.0500	8.89×10^{-7}
0.5	1.3×10^{-3}	7.86×10^{-12}	79.84	3.34	0.0501	5.14×10^{-2}
1.0	0.0	7.76×10^{-12}	79.85	2.23	0.0501	0.150
1.5	3.70×10^{-5}	7.99×10^{-12}	78.94	4.37	0.0503	0.352
2.0	0.0	7.76×10^{-12}	80.31	2.38	0.0499	0.779
2.5	4.23×10^{-5}	7.52×10^{-12}	77.82	0.0	0.0503	0.931

Table 4.7: Simultaneous Estimation of Debye Parameters and Depth. Test 1

% Noise	σ	τ	ϵ_s	ϵ_∞	d	residual
True values:	1.0×10^{-5}	8.1×10^{-12}	80.1	5.5	0.05	
Initial values:	1.0×10^{-3}	10.0×10^{-12}	90.0	3.0	0.1	
0.0	8.2×10^{-4}	8.11×10^{-12}	80.10	5.58	0.0500	5.557×10^{-8}
0.5	1.3×10^{-2}	7.88×10^{-12}	79.84	3.48	0.0501	5.149×10^{-2}
1.0	0.0	7.77×10^{-12}	79.85	2.29	0.0501	0.150
1.5	3.70×10^{-2}	7.99×10^{-12}	78.94	4.33	0.0504	0.352
2.0	8.00×10^{-2}	$\underline{12.36 \times 10^{-12}}$	80.60	<u>41.72</u>	<u>0.119</u>	57.63
2.5	4.23×10^{-5}	7.52×10^{-12}	78.82	0.0	0.0504	0.931

Table 4.8: Simultaneous Estimation. Test 2

4.3 Forward or Simulation Problem with the Lorentz Model

The underlying numerical formulation for simulations using the Lorentz model is the same as that for the Debye model up to and including the the finite dimensional system of differential equations in (4.4) and the matrix definition in (4.5). We depart from the previous formulation in the choice of constitutive law to govern polarization. We use the Lorentz equation discussed in Section 2.1. Applying the same scaling as before, we obtain the scaled Lorentz polarization law

$$\ddot{P} + 2\lambda\dot{P} + \hat{\omega}_0^2 P = \hat{\omega}_p^2 E \tag{4.19}$$

where $\hat{\omega}_0 = \omega_0/c, \hat{\omega}_p = \omega_p/c$ and $\lambda = 1/2c\tau$. The derivation of the finite element equations for the polarization precedes exactly as in the Debye case, with the same provision that components of the variables p and $\dot{p}$ which are identically zero are dropped. (These are the ones corresponding to the exterior of the material domain Ω). The resulting system of equations is

$$\begin{aligned} (M + M_\Omega(\epsilon_\infty - 1))\ddot{e} + M_\Omega\ddot{p} & \\ + (\eta_0\sigma M_\Omega + B)\dot{e} + Ke &= \eta_0\mathcal{J} && (4.20)\\ \ddot{p} + 2\lambda\dot{p} + \hat{\omega}_0^2 p - \hat{\omega}_p^2 e &= 0. && (4.21) \end{aligned}$$

By substituting equation (4.21) into equation (4.20) we obtain an equivalent system of equations

$$\begin{aligned} (M + (\epsilon_\infty - 1)M_\Omega)\,\ddot{e} + (\eta_0\sigma M_\Omega + B)\dot{e} & \\ + (\hat{\omega}_p^2 M_\Omega + K)e - \hat{\omega}_0^2 M_\Omega p - 2\lambda^2 M_\Omega\dot{p} &= \eta_0\mathcal{J} && (4.22)\\ \ddot{p} + 2\lambda\dot{p} + \hat{\omega}_0^2 p - \hat{\omega}_p^2 e &= 0. && (4.23) \end{aligned}$$

This can also be written as a first order system in the composite variable $x = (e, p, \dot{e}, \dot{p})$ as

$$\bar{M}\dot{x} + \bar{K}x = F$$

where

$$\bar{M} = \begin{bmatrix} I & & & \\ & I & & \\ & & M_r & \\ & & & I \end{bmatrix} \tag{4.24}$$

$$\bar{K} = \begin{bmatrix} 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \\ K_1 & K_2 & K_3 & K_4 \\ -\hat{\omega}_p^2 I & \hat{\omega}_0^2 I & 0 & 2\lambda I \end{bmatrix}$$

and the submatrices are

$$\begin{aligned} M_r &= M + (\epsilon_\infty - 1) M_\Omega \\ K_1 &= (\hat{\omega}_p^2 M_\Omega + K) \\ K_2 &= -\hat{\omega}_0^2 M_\Omega \\ K_3 &= (\eta_0 \sigma M_\Omega + B) \\ K_4 &= -2\lambda^2 M_\Omega \end{aligned} \tag{4.25}$$

In the implementation, a matrix-free method is used which avoids constructing the large and irregularly sparse matrices $\bar{M}$ and $\bar{K}$. The matrix-free implementation works by providing a function which computes the derivative of the composite state variable x to a software routine which performs the quadrature. The function for computing the derivative is made faster by re-using the computation of $\ddot{p}$, which is part the of the derivative of the state and also appears in equation (4.20) where it is used in the direct computation of $\ddot{e}$.

Difficulties arise in the computation of solutions to the above equation when using physically realistic values of the material parameters, as the resulting linear system of equations is exceedingly stiff. We use the parameter values $\epsilon_s = 2.25, \epsilon_\infty = 1.0, \omega_0 = 4.00 \times 10^{16}, \tau = 3.57 \times 10^{-16}$ and the corresponding plasma frequency is $\omega_p = 4.472 \times 10^{16}$. These are typical values in the study of physical optics [BF]. We perform the

discretization described above on a grid of size $N = 100$ with the left edge of the material located at $z = 0.01$ and the right edge at $z = 0.1$. We computed the matrices $\bar{M}$ and $\bar{K}$ using Matlab and from this found the condition number of the linear system matrix $\kappa(\bar{M}^{-1}\bar{K}) = 3.23 \times 10^{19}$, making this an extremely stiff system to be integrated. In the fourth order Runga-Kutta methods applied to this equation, the largest step sizes which did not lead to the blowup of the system are generally around $k = 1 \times 10^{-8}$. To generate solutions to this system of equations, we must be content with very short simulations which observe the pulse propagating through a very short distance.

The following results are generated from a simulation where the material occupies the interval $(1.0 \times 10^{-6}, 1.0 \times 10^{-5})$, and with material parameters $\epsilon_s = 2.25, \epsilon_\infty = 1.0, \omega_0 = 1.779 \times 10^{16}, \tau = 7.14 \times 10^{-16}$. The frequency of the interrogation signal is taken to be 8×10^5 GHz and the duration of the signal is equal to twelve of its periods. (The high frequency is necessary in order to see the formation of the Brillouin and Sommerfeld precursors associated with the Lorentz model.) The resulting pulse train thus has twelve complete cycles. Solutions from the simulation are are displayed graphically in Figures 4.13, 4.14, 4.15, and 4.16.

In Figure 4.13 the pulse is traveling to the right after having just entered the material. In Figure 4.14 it has undergone two reflections, first off the supra-conducting back boundary followed by a partial reflection off the surface at z_1. Figure 4.15 is two reflections later again, and in Figure 4.16 the pulse is traveling to the left after another reflection off the back boundary. Here we can see that the pulse has resolved into a part which resembles the Brillouin precursors encountered in simulations of the Debye model, and another part associated with the Sommerfeld precursors. These precursors are examined in detail in [BF] using a frequency domain approach.

These plots were made from a simulation using one thousand basis elements in the discretization of the space variable and with a step size of $k = 3.0 \times 10^{-9}$ in the scaled time variable. This represents an actual time step of 1.0×10^{-8} ns and over 26,000 steps are required to reach the state shown in Figure 4.16. Certain limitations of the

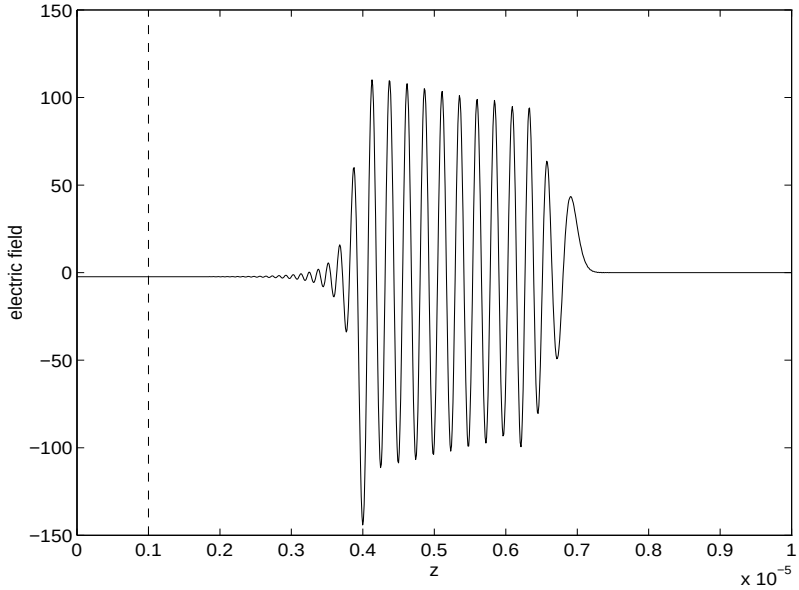

Figure 4.13: Lorentz model simulation ($t = 3.33 \times 10^{-5}$ ns)

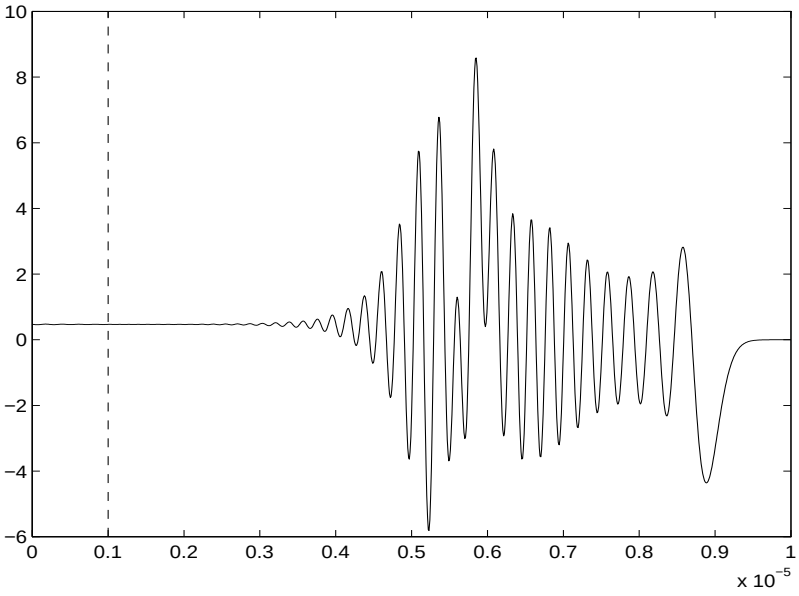

Figure 4.14: Lorentz model simulation ($t = 1.33 \times 10^{-4}$ ns)

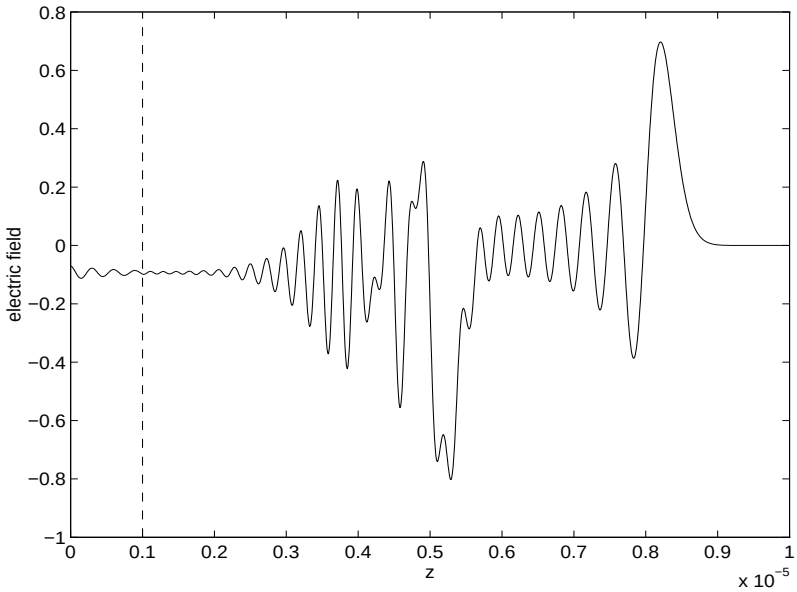

Figure 4.15: Lorentz model simulation ($t = 2.2 \times 10^{-4}$ ns)

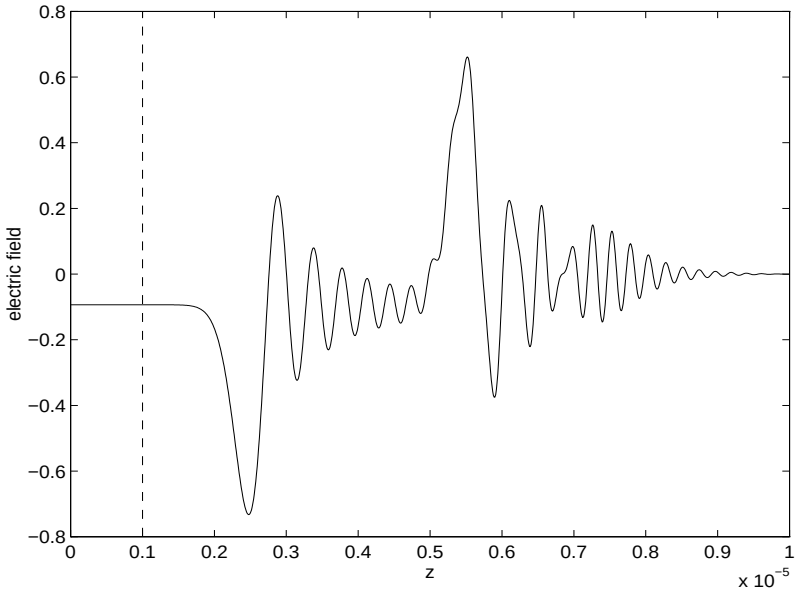

Figure 4.16: Lorentz model simulation ($t = 2.67 \times 10^{-4}$ ns)

accuracy of the simulation are also visible in Figures 4.13 through 4.16. The persistent distance of the solution from zero outside of the pulse is accumulated numerical error. Although this error appears negligible in Figure 4.13, the subsequent decay in the magnitude of the pulse continually increases the relative size of this error.

4.4 The Inverse or Estimation Problem with the Lorentz Model

Because of the long execution times of the forward simulations using the Lorentz model, we restrict our attention to the identification of parameters using the reflection of the interrogating signal off the surface of the material and do not attempt to identify the material's depth. Based on our experience with the Debye equation, we also do not attempt to identify the conductivity parameter σ. However, unlike the Debye problem, this problem is adequately sensitive to the parameter ϵ_∞. Our attempts at estimating the parameters in the Lorentz model take place in a test problem with a slightly different set of physical parameters than used in the example above (since those parameters were chosen to accent the precursor development for demonstration, we now choose parameters more representative of real materials [BF]). Here we use $\epsilon_s = 2.25, \epsilon_\infty = 1.0, \omega_0 = 4.0 \times 10^{16}, \tau = 3.57 \times 10^{-16}$ The interrogating signal is given a frequency of 1.2×10^{14} Hz and the signal is stopped after four complete periods of the input ($\frac{1}{3} \times 10^{-13}$ sec). The data for the inverse problem is depicted graphically in Figure 4.17

The first trials of the inverse problem attempt to establish a rough radius of the acceptable initial values for the parameters to be successfully estimated. This optimization is attempted first over the two parameters ω_0 and ω_p and then over three parameters ω_0, ω_p and τ. Note that the optimization over τ is actually performed through the related parameter $\lambda = 1/2c\tau$. The initial values for the parameters are found by perturbing the "true" values by various relative values, i.e., $\omega_0^0 = (1 + \chi)\omega_0^*$. A value of $\chi = 0.05$ yields a 5% perturbation of ω_0, for example.

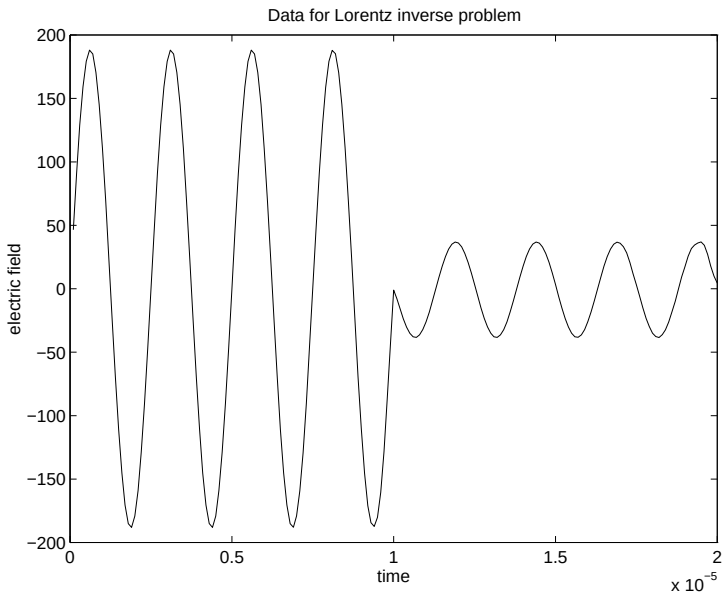

Figure 4.17: Data for Lorentz inverse problem

% Distance	ϵ_∞	ω_0	ω_p	τ	residual
True values:	1.0000	4.0000×10^{16}	4.4721×10^{16}	3.57×10^{-16}	0.0
20%	1.0001	3.9958×10^{16}	4.4675×10^{16}	3.57×10^{-16}	4.311×10^{-2}
30%	1.0001	4.0000×10^{16}	4.4720×10^{16}	3.55×10^{-16}	1.564×10^{-3}

Table 4.9: Convergence results over ω_0 and ω_p

The first attempts on the Lorentz model with the l_∞ trust region code used with the Debye model encountered difficulties arising from the scale and sensitivity to the parameters. The algorithm quickly became saddled with a trust region which was too small to allow sufficient progress to the minimum, or which caused successive steps short enough to trigger the convergence criterion prematurely. This was remedied by switching to a line-search method and scaling the parameters to give them similar magnitudes. The scaling factors used with each parameter are 1 for ϵ_∞, 1×10^8 for ω_0 and ω_∞ and 1×10^6 for λ. This produced the results given in Table 4.9.

% Noise	ϵ_∞	ω_0	ω_p	τ	residual
True values:	1.0	4.000×10^{16}	4.472×10^{16}	3.570×10^{-16}	
Initial values:	1.1	4.400×10^{16}	4.919×10^{16}	3.246×10^{-16}	
1.0%	1.00016	4.006×10^{16}	4.480×10^{16}	3.239×10^{-16}	6.89
2.0%	1.00000	4.019×10^{16}	4.496×10^{16}	3.238×10^{-16}	13.78
3.0%	1.00038	4.028×10^{16}	4.472×10^{16}	3.236×10^{-16}	20.68
4.0%	1.00012	4.034×10^{16}	4.516×10^{16}	3.233×10^{-16}	27.57
5.0%	1.79856	2.540×10^{16}	1.715×10^{16}	3.100×10^{-16}	34.46
6.0%	1.87854	2.356×10^{16}	1.445×10^{16}	2.976×10^{-16}	41.35

Table 4.10: Results of Lorentz Inverse Problem in the Presence of Noise

The Table 4.9 suggests that initial parameter values which are within 30% of the true values are suitable for initial iterates, at least in the absence of noise in the data. This range of parameter values should be compatible with reasonable *a priori* knowledge of the parameters of a real material.

The inverse problem is then attempted in the presence of noise of uniform relative amplitude applied to the observed data in the manner described in Section 4.2 for the Debye problem. The initial values for the parameters are perturbed by 10% from the correct values (with one exception noted below) and the results of these simulations are given in Table 4.10.

We notice that the quality of the results deteriorates at 5.0% noise and that τ is the most difficult of the parameters to recover accurately. When considered as a function of τ only, the objective function exhibits considerable sensitivity with respect to noise, as shown in Figure 4.18. Here, we can see that the objective function is much less sensitive to the value of τ even in the presence of a small amount of noise. The minimizing values of τ in the presence of various amounts of noise is given in Table 4.11.

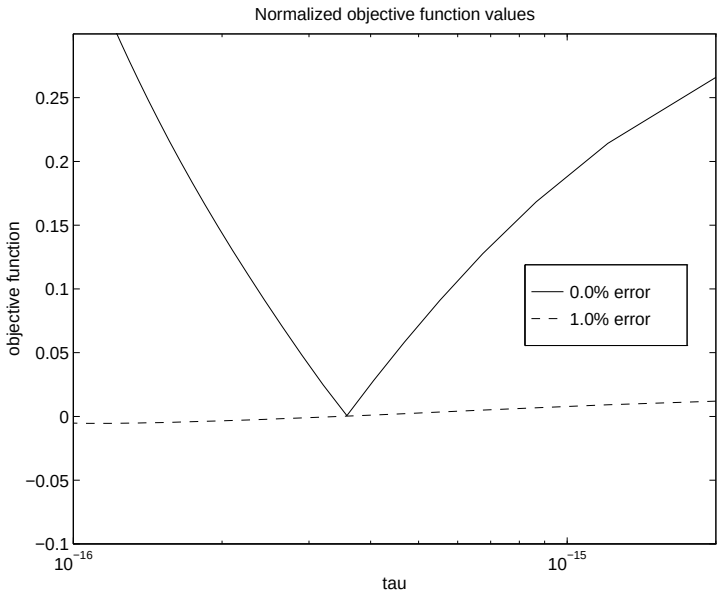

Figure 4.18: Plot of $J(\vec{q}) - J(\vec{q}^*)$ versus τ

% noise	estimated τ	residual
1.0%	1.1291×10^{-16}	6.89
2.0%	6.1779×10^{-17}	13.78
3.0%	4.5353×10^{-17}	20.68
4.0%	3.6834×10^{-17}	27.54
5.0%	3.1432×10^{-17}	34.43
7.0%	2.4687×10^{-17}	48.21

Table 4.11: Results of identifying τ

4.5 Forward or Simulation Problem with General Polarization Models

We begin with the differential equation (2.8) and develop a variational form of the problem much as described in Section 4.1. The time and polarization variables are scaled in the same fashion, but we do not apply the method of mappings to scale the space variable. In order to model materials of various thicknesses, we allow the computational domain to be of arbitrary size, i.e., $\tilde{\Omega} = [0, l]$ for $l > 0$. (The right boundary of the material domain is also not required to be $z_2 = l$ but must coincide with one of the nodes in the spatial discretization. Hence, $z_2 = \bar{z}_R^N, \Omega = [\bar{z}_L^N, \bar{z}_R^N]$, and $\Omega_0 = \tilde{\Omega} - \Omega$. For our examples we will always take $z_2 = l$.) We allow a more general problem by considering both the supraconductive boundary condition at $z = l$ or allowing an absorbing boundary condition like the one at $z = 0$. Although it is not a generally used part of these models, we also retain the possibility of instantaneous polarization. Hence, $\epsilon_r \geq 1$. Also, we will assume a zero initial condition for the electric field.

Our more general weak form of the equation is

$$\begin{aligned}&\langle \mu_0 \epsilon \ddot{E}, \phi \rangle + \langle \mu_0 \sigma \dot{E}, \phi \rangle + \langle \mu_0 \ddot{P}, \phi \rangle \\ &\quad + \langle E', \phi' \rangle + \frac{1}{c} \dot{E}(t, 0)\phi(0) + \frac{1}{c} \dot{E}(t, l)\phi(l) = -\langle \mu_0 \dot{J}_s(t, \cdot), \phi \rangle\end{aligned}$$

The new term is a contribution of the absorbing boundary condition at $z = 1$ and the inner product has been redefined as $\langle \psi, \phi \rangle = \int_0^l \psi(z)\phi(z)dz$.

The Galerkin finite element approximation proceeds as in Section 4.1. Our two cases for the boundary condition at $z = l$ are implemented by either retaining or omitting the function $\phi_N^N(z)$ from the finite dimensional space of test functions. Let $\bar{N}$ denote the index of the highest test function included and $V^{\bar{N}} = \text{span}\{\phi_0^N, \phi_1^N, \ldots, \phi_{\bar{N}}^N\}$ be the space spanned by these functions. The supraconductive boundary is implemented with $\bar{N} = N - 1$ while the absorbing boundary condition is implemented with $\bar{N} = N$.

For polarization, we use the weak form of equation (2.12). Applying the same scaling as before ($\tilde{t} = ct, \tilde{P} = P/\epsilon_0$) and using the approximation of E in equation (4.2) we obtain the following variational equations:

$$\sum_{i=0}^{\bar{N}} \ddot{e}_i(t)\langle \epsilon_r \phi_i, \phi\rangle + \langle \int_{-t}^{0} \alpha(s,\cdot) \sum_{i=0}^{\bar{N}} \dot{e}_i(t+s)\phi_i ds, \phi\rangle + \sum_{i=0}^{\bar{N}} \dot{e}_i(t)\langle \gamma\phi_i, \phi\rangle + \sum_{i=0}^{\bar{N}} e_i(t)\langle \phi_i', \phi'\rangle + \sum_{i=0}^{\bar{N}} \dot{e}_i(t)\phi_i\phi\Big|_{z=0,l} = -\eta_0\langle \dot{J}_s, \phi\rangle \tag{4.26}$$

We find the variational form of the equation by allowing $\phi = \phi_j$ for ϕ_j in $V^{\bar{N}}$, obtaining a system of size $\bar{N}+1$. For a function $\alpha(t,z)$ which varies in both variables, the integral term becomes:

$$\sum_i \int_{-t}^{0} \dot{e}_i(t+s)\langle \alpha(s)\phi_i, \phi_j\rangle ds$$

We make the following definitions:

$$\begin{aligned} M^r_{ij} &= \langle \epsilon_r\phi_i, \phi_j\rangle \\ \hat{M}_{ij} &= \langle I_\Omega(\eta_0\sigma(\cdot) + g(0,\cdot))\phi_i, \phi_j\rangle + \phi_i\phi_j|_{z=0,l} \\ K_{ij} &= \langle \phi_i', \phi_j'\rangle \\ A(t)_{ij} &= \langle I_\Omega \alpha(t,\cdot)\phi_i, \phi_j\rangle \end{aligned} \tag{4.27}$$

Note that $A(t) = [A(t)]_{ij}$ is a time-varying matrix-valued function, while $M^r, \hat{M}$ and K are constant matrices of size $\bar{N}$. We can now write our system as:

$$M^r\ddot{e} + \hat{M}\dot{e} + \int_{-t}^{0} A(s)\dot{e}(t+s)ds + Ke = f(t) \tag{4.28}$$

and $f(t) = -\eta_0\dot{J}_s(t)\hat{e}_1$ where $\hat{e}_1$ is the unit vector with 1 in the first component.

4.5.1 Galerkin Methods for the History Approximation

We choose a method for approximating the history of $\dot{e}(t)$ which maintains the same order of accuracy as the discretization of the rest of the problem. This is because

the order of the accuracy of the solution is limited by the lowest order in accuracy of each of its parts. To do this, we use a Galerkin discretization of the history variable on a finite domain with a degree matching the degree of the discretization in the space variable z. For all of the results presented here, we use degree one elements for each approximation. We can limit the history variable to a finite domain because the hysteresis models of interest to us are all fading memory models, i.e., the kernel function α decays to zero as $s \to -\infty$. The integral over the history of the electric field with this kernel function can be well approximated by truncating it to a sufficiently long interval.

We consider an abstract problem with the appropriate conditions to approximate the history. Let $u(t,s)$ denote the solution to this problem and represent the truncated history of $\dot{e}$ with finite duration r via $u(t,s) = \dot{e}(t+s)$ for $(t,s) \in [0,\infty) \times [-r,0]$. By its relationship to $\dot{e}$, we see that $u(t,s)$ satisfies,

$$\frac{d}{dt}u(t,s) = \frac{d}{ds}u(t,s) \tag{4.29}$$

The appropriate initial condition is $u(0,s) = 0$ since $\dot{e}(s) = 0$ for all $s < 0$. For boundary conditions, we suppose that the state variable $\dot{e}(t)$ is known at all times t, and we can thus impose the boundary condition $u(t,0) = \dot{e}(t)$. Since it is a first order hyperbolic problem, there is a unique solution specified by $u(t,s) = 0$ when $t < -s$ and $u(t,s) = \dot{e}(t+s)$ when $t > -s$.

We create a Galerkin finite element approximation in the history variable s, much as done for the space variable z in Section 4.1. Let ψ_i^M be continuous basis functions defined on the interval $[-r,0]$ with inner product $\langle \psi_i, \psi_j \rangle = \int_{-r}^{0} \psi_i(s)\psi_j(s)ds$. There are a number of choices one might make for these elements, e.g., piecewise linear, quadratic, or higher order polynomial elements. Since our space variable approximations were made with piecewise linear elements, we chose the same type of elements here for compatibility of order. These functions are defined on the uniform grid $\bar{s}_i = -r + \frac{r}{M}i$ for $i = 0, \ldots, M$ where $\psi_i(\bar{s}_j) = \delta_{ij}$.

The past history of $\dot{e}$ is approximated as an element of the space spanned by these

basis functions:

$$u(t,s) \approx u^M(t,s) = \sum_{i=0}^{M} u_i(t)\psi_i(s). \tag{4.30}$$

Recall that $\dot{e}$ is a vector of size $\bar{N}$, and so each of the u_i are vectors of the same size. Let the matrix U denote these unknowns taken together as rows:

$$\begin{aligned} U(t) &= \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_M(t) \end{bmatrix} \\ &= \begin{bmatrix} u_0^{(1)}(t) & u_0^{(2)}(t) & \cdots & u_0^{(\bar{N})}(t) \\ u_1^{(1)}(t) & u_1^{(2)}(t) & \cdots & u_1^{(\bar{N})}(t) \\ \vdots & \vdots & \ddots & \vdots \\ u_M^{(1)}(t) & u_M^{(2)}(t) & \cdots & u_M^{(\bar{N})}(t) \end{bmatrix} \end{aligned} \tag{4.31}$$

Storing the coefficients u_i as rows in this matrix means that the operators we will derive for the history-evolution are applied on the *left* side of U. This was deliberately chosen to simplify the numerical implementation, since one of these operators needs to be inverted, and the available linear-algebra routines work with matrices applied on the left. Matrices representing operators in the space variable will be applied to the right side of U or to the left side of U^T.

4.5.2 Approximating the history of $\dot{e}(t)$

We consider two different methods for finding approximate solutions to equation (4.29) which differ in their variational form of the expression and how the boundary condition is imposed. The first method is to use a variational form which implies no boundary conditions, consider its resulting finite dimensional approximation, and then use the approximate boundary condition to eliminate an equation and unknown from the system. We will see that this approach has a particularly simple implementation. The second is to write a variational form of the equation which imposes the

boundary condition in a weak sense. Care must be taken in this approach not to introduce a spurious boundary condition at the other boundary $s = -r$.

Imposing $\mathbf{u(t,0) = \dot{e}(t)}$ exactly:

Consider the trivial variational form:

$$\langle u_t, \psi \rangle = \langle u_s, \psi \rangle \tag{4.32}$$

No boundary conditions are implied in a variational sense by this equation, since no integration by parts is performed in order to relate it to the original PDE (4.29). We create a finite-dimensional version by restricting ψ to members of the basis set ψ_i in equation (4.32) and using the approximation (4.30) The following system of equations results:

$$\mathcal{M}\dot{U} = WU \tag{4.33}$$

where $\mathcal{M}_{ij} = \langle \psi_i, \psi_j \rangle$ and $W_{ij} = \langle \dot{\psi}_i, \psi_j \rangle$.

This system is determined in the sense that the derivative is well-defined, but it is not clear which PDE it approximates, since it has nothing to say about the boundary condition necessary in a well-posed continuous problem. We consider the system of equations (4.33) too large by one equation and one unknown, which we eliminate with the auxiliary condition arising from the boundary data. Using the same approximation (4.30) for $u(t,s)$ we find,

$$\sum_{i=0}^{M} u_i(t)\psi_i(0) = \dot{e}(t). \tag{4.34}$$

We assume that $\psi_M(0) \neq 0$ (which is the case for the piecewise linear spline approximations used here) and solve for the coefficient $u_M(t)$. We obtain

$$u_M(t) = \frac{1}{\psi_M(0)} \left(\dot{e}(t) - \sum_{i=0}^{M-1} u_i(t)\psi_i(0) \right). \tag{4.35}$$

For the elements we use in the numerical simulations, this reduces to $u_M(t) = \dot{e}(t)$ because $\psi_i(0) = \delta_{0i}$.

We seek a simple way of implementing the removal of the coefficient $u_M(t)$ from the system of equations (4.33). We can use Gaussian row-reduction to eliminate $\dot{u}_M$ from the left-hand side, and then substitute the expression (4.35) wherever $u_M(t)$ appears on the right. This produces a smaller system of the form: $\bar{\mathcal{M}}\dot{\bar{U}} = \bar{W}\bar{U} + v\dot{e}(t)$, where v is a vector of coefficients.

We avoid this process by noting that the same process of Gaussian operations can be applied just as well to both sides of equation (4.33) after performing the matrix-vector multiplication, as they can to the matrix system itself. This allows us to substitute the value of $u_M(t)$ in equation (4.35) before the elimination operations.This yields the following procedure for evaluating the derivative of the reduced system (which is missing $u_M(t)$).

1. Compute the correct boundary value of $u_M(t)$ from (4.35)
2. Compute $\dot{U}$ from equation (4.33) using the entire state $u_i, i = 1, \ldots, M$.

This also produces a value for $\dot{u}_M(t)$ which we ignore as the evolution of this term is determined by the boundary condition.

Another possible advantage of this method is that the boundary condition is satisfied exactly, instead of in a weak sense. The effects of this on the quality of the rest of the approximation are not known. Since the above method departs from the standard finite-element approach, we can no longer say that our approximation of u is the least-squares projection into the finite-dimensional subspace. For this reason we consider another approximation scheme which imposes the boundary condition as part of the weak form.

Imposing the Condition $\mathbf{u(t,0) = \dot{e}(t)}$ weakly:

As the variational formulation of the problem we choose,

$$\langle u_t, \psi \rangle = -\langle u, \psi_s \rangle - \psi(-r)u(-r) + \dot{e}(t)\psi(0) \tag{4.36}$$

This implies that $u_t = u_s$ and $u(0) = \dot{e}(0)$ under the standard variational arguments. Notice that the term $-\psi(-r)u(-r)$ cancels a term which would otherwise yield the condition $u(-r) = 0$ after integration by parts.

In the usual manner, we find a system of differential equations from the variational form (4.36).

$$\sum_{i=0}^{M} \dot{u}_i(t)\langle\psi_i, \psi_j\rangle = -\sum_{i=0}^{M} u_i(t)\langle\psi_i, \dot{\psi}_j\rangle \\ -\sum_{i=0}^{M} u_i(t)\psi_i(-r)\psi_j(-r) + \dot{e}(t)\psi_j(0). \tag{4.37}$$

We define the matrices $\mathcal{M}, \mathcal{W}$ by $\mathcal{M}_{ij} = \langle\psi_i, \psi_j\rangle$ and $\mathcal{W}_{ij} = \langle\psi_i, \dot{\psi}_j\rangle + \psi_i(-r)\psi_j(-r)$, then express the system of differential equations (4.37) in matrix form as

$$\mathcal{M}\dot{U}(t) = -\mathcal{W}U(t) + \Psi\dot{e}(t)^T \tag{4.38}$$

where $\Psi(0) = (\psi_0(0), \psi_1(0), \ldots, \psi_M(0))^T$. We note that the $M+1$ rows of U are vector valued functions in $\mathbb{R}^{1\times\bar{N}+1}$. This differs from the convention of treating e and $\dot{e}$ as column vectors, hence, the appearance of $\dot{e}^T$ in equation (4.38).

The matrix equation (4.38) is appropriate for implementation, however we seek to simplify the implementation further. Through integration by parts we find

$$\begin{aligned} \mathcal{W}_{ij} &= \langle\psi_i, \dot{\psi}_j\rangle + \psi_i(-r)\psi_j(-r) \\ &= -\langle\dot{\psi}_i, \psi_j\rangle + \psi_i\psi_j|_{-r}^{0} + \psi_i(-r)\psi_j(-r) \\ &= -W_{ij} + \psi_i(0)\psi_j(0) \\ &= -W_{ij} + \Psi\cdot\Psi^T \end{aligned}$$

where $W_{ij} = \left\langle\dot{\psi}_i, \psi_j\right\rangle$ and equation (4.38) becomes:

$$\begin{aligned} \mathcal{M}\dot{U} &= WU - \Psi\Psi^T U + \Psi\dot{e}(t)^T \\ &= \hat{W}U + \Psi\dot{e}(t)^T \end{aligned}$$

where $\hat{W}_{ij} = W_{ij} - \psi_i(0)\psi_j(0)$. This equation can also be written as: $\mathcal{M}\dot{U} =$

$WU + \Psi(\dot{e}(t)^T - \Psi^T U)$. Here we can more clearly see the presence of the boundary condition in the weak form. The boundary condition can itself be expressed as $\dot{e}(t)^T = \sum_{i=1}^{M} \psi_i(0) u_i = \Psi^T U$.

Note that the time varying terms in equation (4.38) are very few provided that for only a few i, $\psi_i(0) \neq 0$. For the standard first-order splines we use, $\psi_i(0) = \delta_{iM}$ so only one element is non-zero.

4.5.3 Implementing the Hysteresis Term

We express the integral term in equation (4.28) in terms of the Galerkin approximation. We have

$$\begin{aligned}
\int_{-t}^{0} A(s)\dot{e}(t+s)ds &\approx \int_{-r}^{0} A(s)u(t,s)^T ds \\
&\approx \int_{-r}^{0} A(s)u^M(t,s)^T ds \\
&= \int_{-r}^{0} A(s) \sum_{j=0}^{M} u_j^T(t)\psi_j(s) ds \\
&= \sum_{j=0}^{M} \int_{-r}^{0} A(s)\psi_j(s) ds \, u_j^T(t) \\
&= \sum_{j=0}^{M} A^j u_j^T(t) \qquad (4.39)
\end{aligned}$$

where $A^j = \int_{-r}^{0} A(s)\psi_j(s)ds$. Recall that $A(s)$ is a matrix function, and so the A^k are matrices of size $\bar{N}+1 \times \bar{N}+1$, where the ij element of A^k is:

$$A_{ij}^k = \int_{-r}^{0} \langle A(s,\cdot)\phi_i, \phi_j \rangle \psi_k(s) ds. \qquad (4.40)$$

This gives us the complete discretization of the history terms in the second order system of equations (4.28), which we now can write as:

$$\begin{aligned}
M^r \ddot{e} + \hat{M}\dot{e} + \sum_{k=0}^{M} A^k u_k^T + Ke &= f(t) \\
\mathcal{M}\dot{U} + \mathcal{W}U &= \Psi \dot{e}^T
\end{aligned}$$

where $f(t) = -\eta_0 \dot{J}_s(t)\hat{e}_1$ and $\Psi = (\psi_1(0), \psi_2(0), \ldots, \psi_M(0))^T$. We convert this system to first order by letting $x = e$, $y = \dot{e}$.

$$\begin{aligned} \dot{x} &= y \\ M^r \dot{y} &= -\hat{M}y - Kx - \sum_{k=0}^{M} A^k u_k^T + f(t) \qquad (4.41) \\ \mathcal{M}\dot{U} &= -\mathcal{W}U + \Psi y^T. \qquad (4.42) \end{aligned}$$

The state variables of the system we need to solve are the length $\bar{N}+1$ vectors x,y and the $M+1 \times \bar{N}+1$ matrix U. The matrices A^k, K, M, and $\hat{M}$ are as defined in equations (4.27). The matrices $\mathcal{M}$ and $\mathcal{W}$ and the vector Ψ depend only on the basis functions chosen for the history approximation, while $M^r, \hat{M}$ and the A^ks depend on physical parameters we will want to estimate.

4.5.4 Specific Implementation: Constant Material Parameters

We consider the special case of materials which are homogeneous throughout the domain Ω. Functions which describe dielectric properties of these materials are constants modulated by the indicator function of the domain Ω. These are given by

$$\begin{aligned} \sigma(z) &= \sigma I_\Omega(z) \\ \epsilon_r(z) &= 1 + I_\Omega(z)(\epsilon_r - 1) \qquad (4.43) \\ \alpha(s,z) &= \alpha(s) I_\Omega(z) \\ g(0,z) &= g_0 I_\Omega(z) \end{aligned}$$

We have introduced new parameters σ and g_0 for the conductivity and zero value of g, and the kernel function $\alpha(s)$.

As in Section (4.1), we further assume that the boundaries of the material coincide with the domains which define the test functions. This way, the material parameters are constant over each domain over which an integral is evaluated. The matrices

in (4.27) become

$$\begin{aligned} M^r &= M + (\epsilon_r - 1)M_\Omega \\ \hat{M} &= (\eta_0\sigma + g_0)M_\Omega + B \\ A(s) &= \alpha(s)M_\Omega \end{aligned}$$

where

$$\begin{aligned} M_{ij} &= \langle \phi_{i-1}, \phi_{j-1} \rangle \\ M_{\Omega ij} &= \langle I_\Omega \phi_{i-1}, \phi_{j-1} \rangle \quad \text{and} \\ B_{ij} &= \phi_{i-1}(0)\phi_{j-1}(0) + \phi_{i-1}(1)\phi_{j-1}(1), \end{aligned}$$

representing the contributions of the boundary conditions. For the case of a supracondicting boundary at $z = 1$ then $B_{ij} = \phi_{i-1}(0)\phi_{j-1}(0)$ and the space of test functions is reduced to $V_R^N = \text{span}\{\phi_0^N, \ldots, \phi_{N-1}^N\}$.

representing the contributions of the boundary conditions. For the case of a supracondicting boundary at $z = 1$ then $B_{ij} = \phi_{i-1}(0)\phi_{j-1}(0)$ and the space of test functions is reduced to $V_R^N = \text{span}\{\phi_0^N, \ldots, \phi_{N-1}^N\}$.

We use equation (4.40) to find the specific form of the matrices A^k:

$$\begin{aligned} A_{ij}^k &= \int_{-r}^0 \langle \alpha(s,\cdot)\phi_i, \phi_j \rangle \psi_k(s) ds \\ &= \int_{-r}^0 \alpha(s) M_{\Omega ij} \psi_k(s) ds \\ &= M_{\Omega ij} \int_{-r}^0 \alpha(s)\psi_k(s) ds \end{aligned}$$

So

$$A^k = M_\Omega \alpha_k$$

where $\alpha_k = \int_{-r}^0 \alpha(s)\psi_k(s)ds$. Letting $\alpha = (\alpha_0, \ldots, \alpha_M)^T$ we can express the approximation of the hysteresis approximation in equation (4.39) as

$$\int_{-r}^0 A(s)\dot{e}(t+s)ds = \int_{-r}^0 A(s)u(t,s)^T ds$$

$$\begin{aligned} &\approx \sum_{j=0}^{M} \alpha^j u_j^T(t) \\ &= \sum_{j=0}^{M} M_\Omega \alpha_j u_j^T \\ &= M_\Omega U^T \alpha. \end{aligned}$$

The full system of equations (4.41) is now:

$$\begin{aligned} \dot{x} &= y \\ (M + M_\Omega(\epsilon_r - 1))\dot{y} &= -((\eta_0\sigma + g_0)M_\Omega + B)y - Kx - M_\Omega U^T\alpha + f(t) \\ \mathcal{M}\dot{U} &= -\mathcal{W}U + y\Psi^T \end{aligned} \tag{4.44}$$

where the last equation may still be replaced with:

$$\mathcal{M}\dot{U} = \hat{W}U + \Psi y^T.$$

4.6 Results of Simulations with the General Model

4.6.1 Hysteresis Representation of Debye Model

We measure the accuracy of solutions to the general polarization model by comparing them with those obtained from the differential Debye model of Section 4.1. The correct function to be used in the general polarization model to represent a Debye material is found from the variation of constants solution to the differential equation.

In the scaled version of the equations we have:

$$\dot{P} + \psi P = \epsilon_d \lambda E.$$

Use of an integrating factor yields:

$$P(t) = \int_0^t g(t-\xi)E(\xi)d\xi$$

where $g(s) = \epsilon_d \lambda e^{-\lambda s}$. From this we calculate:

$$\ddot{P}(t) = \int_{-t}^{0} \dot{g}(-s)\dot{E}(t+s)ds + g(0)\dot{E}(t). \tag{4.45}$$

r	M	Δs	Max Difference	% Difference	execution time
0.02	5	4.0×10^{-3}	0.4038	1.04	36.08
0.02	7	2.86×10^{-3}	0.5705	1.47	40.27
0.02	10	2.0×10^{-3}	0.6824	1.76	46.09
0.02	20	1.0×10^{-3}	0.6662	1.73	68.29
0.03	7	4.28×10^{-3}	0.4011	1.04	40.13
0.03	10	3.0×10^{-3}	0.1119	0.29	45.79
0.03	15	2.0×10^{-3}	0.0149	3.84×10^{-2}	56.3
0.03	30	1.0×10^{-3}	0.0120	3.10×10^{-2}	91.13
0.04	10	4.0×10^{-3}	0.3101	0.801	45.92
0.04	13	3.08×10^{-3}	0.1117	0.289	51.93
0.04	20	2.0×10^{-3}	0.0223	5.94×10^{-2}	67.75
0.04	40	1.0×10^{-3}	0.0016	4.24×10^{-3}	113.44

Table 4.12: Comparison of General and Differential Debye Simulations

The function α which appears in the definitions (4.43) is $\alpha(s) = -\epsilon_d \lambda^2 e^{\lambda s}$ and the remaining parameters are: $\epsilon_r = \epsilon_\infty$, $g_0 = \epsilon_d \lambda$.

We compute the solution to a test problem using both the differential and hysteresis Debye polarization models. The results are summarized in Table 4.6.1. The physical parameters are $\sigma = 0.0, \epsilon_\infty = 5.5, \epsilon_s = 80.1$ and $\tau = 8.1 \times 10^{-12}$ and the simulation is run to $t = 1.67 \times 10^{-9}$. The boundaries of the material medium are $z_1 = 0.005$ and $z_2 = 0.1$. The discretization in space is a first-order finite element approximation with $N = 100$ in for both the differential and hysteresis models. The hysteresis simulations also use a first-order finite element approximation in the history variable. That is, the functions ϕ_i and ψ_i are both piecewise linear functions defined on discretizations of the intervals $z \in [0, l]$ and $s \in [-r, 0]$ respectively.

The maximum difference is computed as $\|e_d - e_h\|_\infty$ and the percent difference is $\|e_d - e_h\|_\infty / \|e_d\|_\infty$, where e_d and e_h are the solutions obtained from the differential and hysteresis models respectively. The last column is an average value of the actual computer time in seconds taken to solve the test problem. These figures should be contrasted with the 6.65 seconds time for the differential Debye simulation using the

same discretization in the space variable. We see that agreement between the two methods is very good when sufficiently accurate discretizations of the history variable are used. This requires that a sufficiently long history be retained in the simulation (as measured by r) and a sufficiently fine discretization of this interval (measured by $\Delta s = r/M$). The results show that a history duration of at least $r = 0.03$ (about 10 ns, in the unscaled time variable) is necessary for results with small relative error.

Comparing results from the table, we see that similar results are obtained by $M = 10, r = 0.03$ and $M = 13, r = 0.04$ while the computation with $M = 13$ takes about 14% longer to execute. Although more accurate results are obtained for higher values of M with $r = 0.04$ than with $r = 0.03$, these are only incremental improvements in the quality of the simulation. Since we are interested in the inverse problem which requires repeated simulations of the system, the time of execution is a primary concern.

We show graphically in Figure 4.19 the close fit of results generated with the two formulations (i.e., differential Debye versus hysteretic Debye polarization models). This plot is for $M = 4$ and $r = 0.03$ in the hysteresis formulation. For more accurate simulations the graphs are indistinguishable in the plot. Figure 4.20 exhibits a case where an insufficiently refined discretization of the history variable leads to a very poor approximation. This result was computed with $M = 3$ and $r = 0.03$.

4.6.2 Hysteresis Representation of Lorentz Model

The representation of the Lorentz model of polarization is plagued by computational difficulties which make it impractical for simulations.

The plot of the hysteresis kernel using the parameter values used in the inverse problem of Section 4.4 is given in Figure 4.21. This function is the derivative of the impulse response function which appears in the variation of constants solution to the differential equation. Because of the oscillatory nature of the function, a much finer approximation of the history variable is required, making the size of the resulting system of equations prohibitively large. For example, the plot of the kernel

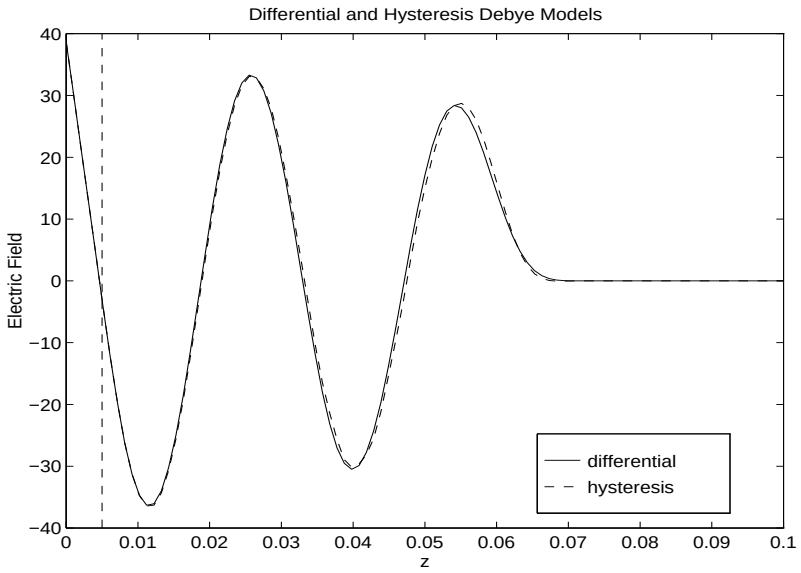

Figure 4.19: Results from Differential and Hysteresis Debye Models

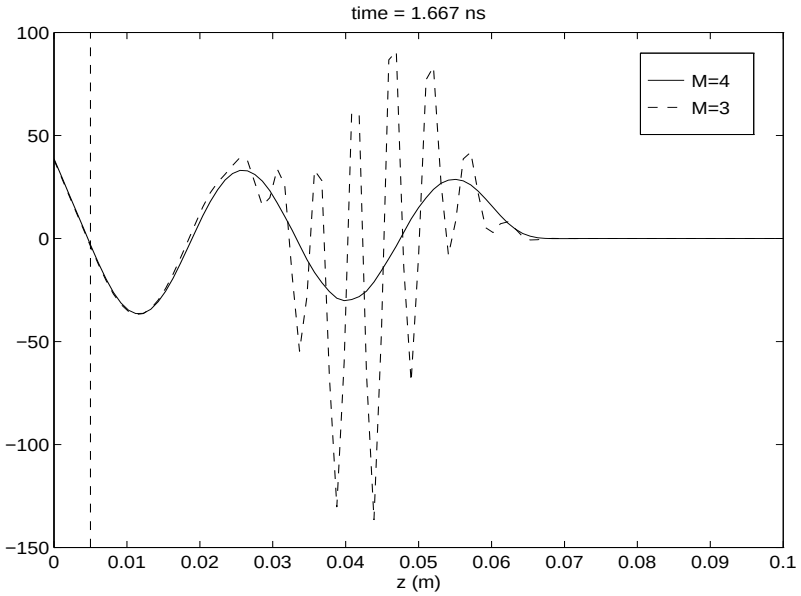

Figure 4.20: Breakdown of Hysteresis Debye Model

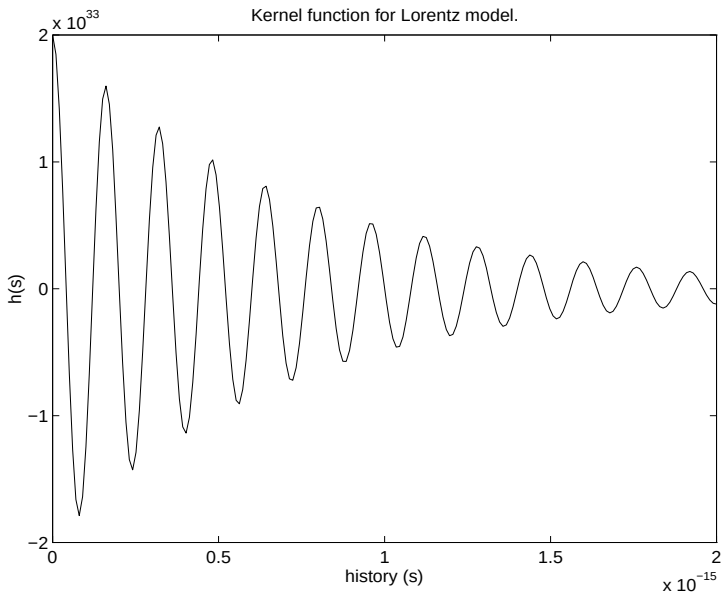

Figure 4.21: Hysteresis kernel for Lorentz model with $\tau = 3.57 \times 10^{-16}$

function in Figure 4.21 suggests that a history duration of $r = 2 \times 10^{-15}$ is necessary before the kernel function drops to a level where it can be neglected. This includes twelve complete oscillations. To approximate this function accurately would require on the order of 36 basis elements in the Galerkin discretization of the history variable, allowing for a meager three elements per cycle.

Furthermore, stability requirements limit the discretization in time of the entire system to being no larger than the discretization of the history variable. This arises from the discretization of the evolution equation of u in equation (4.29). The resulting discretizations in equations (4.33) and (4.38) are subject to a stability condition which limits the size of the discretization in time. Since we do not want the propagation speed of the numerical scheme ($\Delta s/\Delta t$) to be faster than the actual propagation speed of the solutions (1, in the scaled variables), the time step can be no larger than the discretization used in the history variable. In the example considered here, the resulting discretization in the history variable would be approximately $\Delta s =$

5.6×10^{-17}, suggesting a time step of similar size. (The similar differential problem in Section 4.3 required an even smaller step size of 1.0×10^{-17} for an accurate result.) A simulation of duration 6.67×10^{-14} (as used in the inverse problem of Section 4.4) would require about 1200 steps. Because of the large number of basis elements in the history variable, the resulting computation is prohibitively expensive.

4.7 Inverse or Estimation Problem with General Polarization Model

Having successfully duplicated the quantitative behavior of the Debye model of polarization with a hysteretic polarization model, we turn to using this method of simulation in an inverse problem. Different inverse problems can be formulated in this way depending on the way in which the hysteresis kernel depends on the set of parameters being optimized. One option is to express the kernel as a function of the same set of parameters which appear in the differential equation, $\sigma, \epsilon_d, \epsilon_\infty$ and τ and hence in the variation of constants solution. For the Debye differential equation, this function was computed in Section 4.6.1 as $g(s) = \epsilon_d \lambda e^{-\lambda s}$ and the kernel function is $\alpha(s) = -\epsilon_d \lambda^2 e^{\lambda s}$. As noted in Section 4.2 optimizing over the parameter $\lambda = 1/c\tau$ instead of τ in the differential Debye polarization model improved the performance of the optimization problem.

As an alternative to representing the Debye kernel as a function of the Debye physical parameters, we can attempt to reconstruct it directly. From equation (2.12) we see that recovering the hysteresis function $g(t)$ can only be done through identifying a suitable finite dimensional approximation of its derivative α, which appears in the integral term, and its value at zero, which appears in the damping term. Since we're interested in the case where $E(0, z) = 0$, the term κ is zero. Furthermore, in this case, the only appearances of the conductivity σ and the parameter $g(0)$ are both in the damping term. Therefore we do not expect to be able to recover these parameters separately. For the remainder of this problem, we consider $\sigma = 0$ and let $g(0)$ represent

whatever actual conductivity the material may have.

To reconstruct the hysteresis kernel α we attempt to find optimal values for the coefficients in a Galerkin finite element approximation of this function. This Galerkin approximation is performed over a discretization of the history exactly as described in Section 4.5.1. To simplify the computations, the same basis elements are used to approximate the kernel function for the purpose of identification as are used in the simulation. Hence, the coefficients we seek to identify are directly related to the ones used in the forward simulation. Using the notation established in Section 4.5.1, $\mathcal{M}$ is the mass matrix arising from pair-wise inner products of the basis functions: $\mathcal{M}_{ij} = \langle \psi_{i-1}, \psi_{j-1} \rangle$. The values which appear in the forward simulation are $\alpha_k = \int_{-r}^{0} \alpha(s)\psi_k(s)ds$ and the coefficients of the Galerkin approximation of α are given by $h = \mathcal{M}^{-1}\alpha$, where the vector α is $\alpha = \langle \alpha_0, \alpha_1, \ldots, \alpha_M \rangle$ and $h = \langle h_0, h_1, \ldots, h_M \rangle$

We adopt a standard test problem for comparing the relative ease of reconstructing the hysteresis function through the Debye physical parameters and through Galerkin coefficients. The true values for the physical parameters are $\sigma = 0.0$, $\epsilon_\infty = 5.5$, $\epsilon_s = 80.1$ and $\tau = 8.1 \times 10^{-12}$ (Again, these are the typical values for water). The data is gathered from a forward simulation of the problem using the differential Debye polarization model and is depicted graphically in Figure 4.22.

We attempt two inverse problems over different sets of parameters using this test problem. These different sets of parameters arise from the different ways of representing the Debye kernel function, either as a function of the Debye parameters or through the coefficients of a Galerkin approximation. Both of these inverse problems use the result of a forward simulation from the differential Debye model to provide the data. Since no noise is added, we expect to be able to reproduce the parameter values with a high degree of accuracy, depending on the precision of the hysteresis approximation. We do not attempt to identify the parameters σ and ϵ_∞. In Section 4.2 the low sensitivity of the objective function to these parameters was established. Instead, each of these parameters have their values fixed at the value used in generating the data.

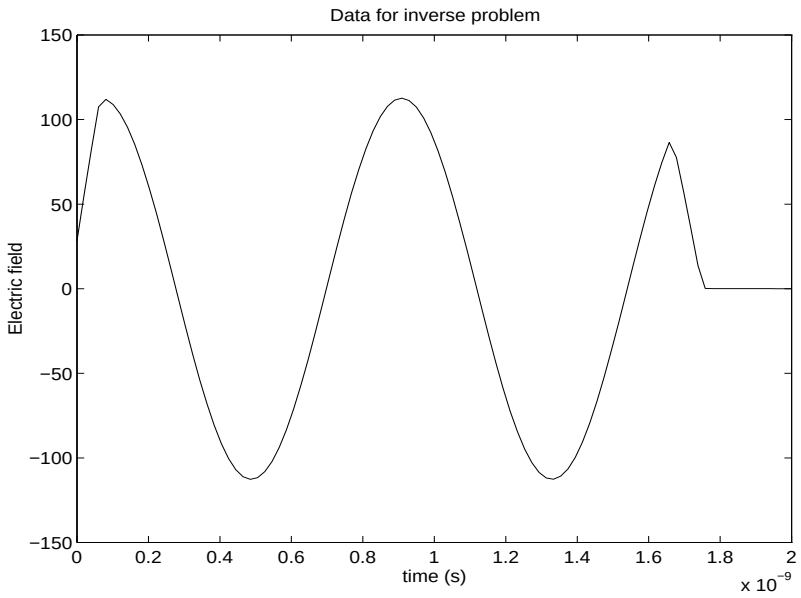

Figure 4.22: Data for Debye Test Problem

The results of fitting the hysteresis Debye model to data generated by the differential Debye model are summarized in Table 4.13. All of these attempts used $r = 0.03$ for the duration of the history approximation. The initial values for the two parameters were $\tau_0 = 7.7142 \times 10^{-12}, \epsilon_{s0} = 83.83$. These results demonstrate that a sufficiently accurate hysteresis implementation of the Debye model yields reasonable estimates for the parameter values in an inverse problem.

The identification of the Galerkin coefficients proved to be considerably more difficult than identification of the Debye model parameters. The "true" values for the parameters are found by taking the vector of coefficients α which are computed to perform the forward simulation and computing the coefficients of the Galerkin approximation h. These true values are then perturbed by 10% to generate the initial values used in the optimization. As summarized in Table 4.14, attempts to identify all of the coefficients h and the value of $g(0)$ generally failed to converge in a reasonable number of iterations. The results for $M = 4$ and $M = 6$ were taken before the

M	estimated ϵ_s	estimated τ	residual
True values:	80.100	8.1000×10^{-12}	
Initial values:	83.8300	7.7142×10^{-12}	
4	81.3481	8.2190×10^{-12}	1.14808
5	79.7209	8.8766×10^{-12}	0.37468
6	80.4458	7.9891×10^{-12}	0.30911
7	79.9598	8.3541×10^{-12}	0.13821
8	80.2243	8.0389×10^{-12}	0.11429
9	80.0418	8.1978×10^{-12}	5.83645×10^{-2}
10	80.1558	8.0660×10^{-12}	5.19723×10^{-2}

Table 4.13: Results of estimating hysteresis Debye model from differential Debye data

program terminated and indicated convergence. (At the time that the values were taken, both programs had been running continuously for at least nine days.) We report the values of the two parameters $g(0)$ and the first coefficient of the Galerkin approximation of the kernel h_0. Since the value h_0 depends on the discretization, its true and initial value both vary with M. These values are summarized in Table 4.15. This result strongly suggests that the problem is over parameterized. To alleviate this problem, we note that h_0 is considerably larger than the other coefficients, owing to the decaying exponential form of the kernel function. This parameter, along with the value of $g(0)$ are likely to have the most influence over the objective function, with the remaining coefficients playing minor roles. We attempt to identify just these two most significant parameters while holding the others fixed at their true values.

This smaller optimization problem also proved to be very difficult, although acceptable results were eventually obtained, as indicated by the results in Table 4.16. We see that this problem required a much larger number of iterations to attain convergence than the two parameter Debye estimation problem summarized in Table 4.13. One possible cause of the very slow convergence is suggested by the objective function. Figure 4.23 is a plot of the logarithm of the objective function over a narrow (5%) range of reach of these parameters. We see that the minimizer lies at the bottom of

M	h_0	$g(0)$	residual	Iterations
True values:	—	2264.20		
Initial values:	—	2490.63		
4*	-683759	2336.54	2.159×10^{-1}	900
5	-765083	2365.97	1.645×10^{-2}	188
6*	-809484	2349.39	1.323×10^{-1}	829

Table 4.14: Results from estimating all coefficients of kernel.

M	True h_0	Initial h_0
4	-640889	-704978
5	-706575	-777233
6	-753628	-828991

Table 4.15: True and initial values of h_0 for various M

M	h_0	$g(0)$	residual	iterations
True Values:	—	2264.20		
Initial Values:	—	2490.63		
4	-640888	2264.20	1.15539×10^{-5}	83
5	-706574	2264.20	1.63708×10^{-5}	80
6	-753625	2264.20	1.98944×10^{-5}	95

Table 4.16: Results of estimating $g(0)$ and h_0

a narrow trench. This is likely caused by the two coefficients g and h_0 being nearly interdependent in their influence on the solution. On one side of the trench we see that the values of the objective function rise slowly, while on the other they increase exponentially fast.

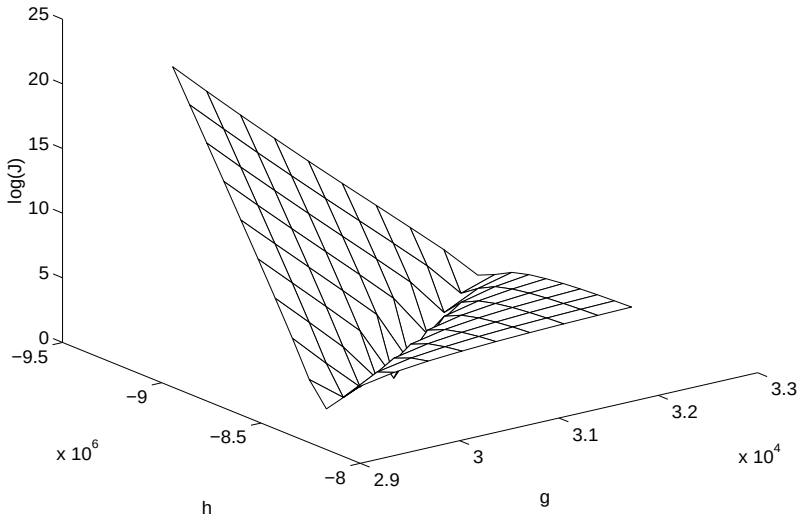

Figure 4.23: $\log J(g(0), h_0)$ for $M = 4$

Chapter 5

Object-Oriented Design and Programming for Scientific Applications

5.1 Introduction

The implementation of computer programs to solve the numerical problems arising in the last chapter have been made part of a broader investigation into the applications of object-oriented programming to scientific computing. The work presented in the next two chapters is guided by two additional objectives; To develop a re-usable library of code which we hope will find application in other research projects, and to investigate the issues of code design which surround the application of object-oriented programming to scientific problems.

C++ is an extension of the widely used programming language C into an object oriented language. We chose C++ for this investigation because of increasing interest in it for scientific computing (see, for example [LAP++, TNT, MTL, Diff]). Furthermore, compilers for C++ are readily available on a variety of platforms.

Our main objective in this project is to implement numerical solutions to the forward and inverse problems described in Chapter 4. Rather than write code which solves these problems exclusively, we concurrently develop code to implement parts of the problem which can also be applied elsewhere. A collection of routines which is

designed for repeated application to various problems is known as a *software library*, or just as a *library*. Developing our library concurrently with its application to these research problems allows us to use these problems as a test case for evaluating the efficiency and usefulness of the library.

We begin by declaring the criteria by which a library of code for scientific programming will be measured. A successful library of code for scientific programming must, to some degree, meet the following three goals:

1. **Re-use:** The library should facilitate the re-use of code. Functions in the library should be written to uniformly apply to a wide range of application. Where possible, the library should also re-use existing code, by providing an interface to already existing and tested routines. When it does so, the library should make such code easier to use.

2. **Clarity:** The library should be conceptually clear and allow the programmer to think on a familiar level of abstraction. Objects in the library should correspond to parts of mathematical problems. The programmer can then work with these more familiar objects instead of the constructs native to the programming language.

3. **Efficiency:** The library should be optimized wherever possible to perform tasks in a minimum of time and using a minimum of memory.

5.1.1 Procedural Programming

We briefly describe how code is organized in non-object-oriented languages. This will highlight the new features of object-oriented programming and its advantages for scientific computing.

Procedural programming is the division of large programming jobs into parts which are implemented as functions or subroutines. These smaller modules of code work together by operating on a common set of data which they receive as *arguments*.

In C these modules are implemented as functions while Fortran uses subroutines for this purpose. When speaking generally, we will refer to these modules of code as *procedures*.

Procedures are usually written to be generic, in the sense of performing tasks which can be useful in a variety of situations and for a variety of data. For example, procedures for factorizing a matrix or sorting an array of data. Procedures can also be written with more specific goals, such as constructing matrices which arise in specific numerical applications. An application program generally is written to compute solutions to a particular problem or class of problems by calling various procedures. Procedures which are written to act generically are the basis of re-usable libraries of code.

5.1.2 The Dual Role of Functions

Throughout various computing languages, the term *function* refers to a module of code which accepts arguments and has a separate mechanism for returning information to the calling program. In Fortran, functions are distinct from *subroutines* which do not return data (except by changing the values of the arguments). In C, and therefore in C++, functions are fundamental to the implementation of procedures. Even the main body of a C or C++ program is a special function with the name `main`. We will see how *classes* are used to further organize code in C++, but the elementary structural unit of code is still the function.

In scientific programming applications, program functions are also used to represent mathematical functions. (This is the primary use of the `function` construct in the Fortran language, since subroutines are used for writing procedures.) We will see in a later section how this implementation of mathematical functions is sometimes too restrictive and how this can be remedied using *objects* in C++.

5.1.3 Pointers and References

The *pointer* is familiar to C programmers as a special data type for holding the address of another data type. For each type in C, there is a corresponding pointer type which can store its address. The name of the pointer type is the same as the name of the type, suffixed with an asterisk, e.g., `int*, double*, char*`. Pointers are indispensable when dealing with arrays of dynamically allocated memory (Section 5.1.4) and in C they are the only method of passing an argument to a function by reference and not by value. Pointers can also be defined which point to functions of a certain type, where the type of a function is determined by the type and number of its arguments and return argument. Pointers require a special syntax for accessing the elements they refer to, called de-referencing the pointer, and is also done with the asterisk. If `data_p` is a pointer of type `double` then `*data_p` is the double precision number stored in that memory location. If the pointer points to the beginning of a block of values, then the other memory positions can be accessed through the `[]` notation. For example `data_p[10]` would point to the 11th double precision number in the block of memory pointed to by `data_p`. (The first number has index 0). The compiler accounts for the different sizes of the various data types when using this notation.

C++ also includes another type which can be thought of as an intermediate between ordinary variables and pointers. A *reference* provides another way of accessing a variable, but uses the same syntax as a regular variable. Reference types are indicated by an ampersand after the type name, e.g., `int&, double&, char&`.

References are the method of choice when passing arguments to a function which we do not wish to be copied. (More on this is in Section 5.3.4). When the argument type in the function is a reference, instead of an ordinary type, the function receives the address of the argument and can access it through the argument variable. For example

```
void myFunction(double& value){
  value = value * 2.0;
}
```

This function accepts a `double&` argument. When it is called, the argument is not copied. Instead the function contains a reference (called `value`) to the variable in the calling code. After execution of the function, the value of the argument passed to `myFunc` will be doubled.

This use of references is even more important when the arguments are *objects* (Section 5.2.1) for which the copying operation is expensive.

5.1.4 Dynamic Memory Allocation

C++ inherits from C the ability to dynamically allocate memory through the computer's operating system. This is an important advantage over older languages like Fortran77 in which arrays all have fixed sizes. C++ programs can request more memory from the operating system and return it when done. This makes variable sized arrays possible, giving programs greater flexibility without having to allocate extra memory.

When a block of memory is granted this way, it is kept track of through a `pointer`, which stores its location in the memory of the computer which stores the address of the first element in the block. The actual location of the memory is rarely of interest to the programmer, so the pointer variable is used indirectly to refer to access the contents of the block. The pointer is also used when the block of memory is returned to the operating system (*de-allocated*) or passed to a function.

Using dynamically allocated memory in C++ creates additional responsibilities. Memory which is no longer used by a program must be returned to the operating system, or it will remain unavailable. The systematic failure to de-allocate memory is known as a *memory leak* and can seriously hamper the performance of a program if free memory becomes scarce. Great care must be taken to ensure that the pointers which keep track of dynamically allocated memory are not lost or have their values changed. (Other object-oriented languages such as Java perform *garbage collection* which means that data is automatically reclaimed by the system when it is no longer in use.)

Like ordinary arrays in C, dynamically allocated arrays do not support range checking. This means it is possible to use a pointer to refer to memory addresses outside its block of memory, possibly into memory being used elsewhere in the program, by other programs, or by the operating system itself. This can result in the program behaving incorrectly, terminating with an error, or even "crashing" the computer if the operating system is affected. We will see in a later section how the use of objects can automate range checking for arrays to prevent these errors.

5.1.5 Operator Overloading

An operator or function is said to be *overloaded* when it has multiple definitions which apply to different arguments. This is familiar to users of Fortran90 (and some versions of Fortran77) in which functions like `sin` are defined to work with numbers of various precisions. The elementary mathematical operators such as `+,-,/,*` are all overloaded to work with the various built-in mathematical types. It is the job of the compiler to deduce which version of the function or operator is intended and apply it correctly.

One difference between C++ and Fortran is that C++ supports defining overloaded functions to work with user defined types. (Note that the various versions of an overloaded function are independent of each other and do not necessarily perform similar operations on their arguments. Creating overloaded functions like this, however, would surely be considered bad design.) The versions of an overloaded function or operator can differ in the type or in the number of their arguments. This is necessary so the compiler can tell which version of the function is intended. It is not possible to overload a function or operator with versions that only differ in the return type.

5.1.6 Inline Functions

Inlining a function is another way that the performance of programs can be enhanced in C++. When a function is *inlined* by the compiler, the body of the function is substituted into other code where the function is called. This avoids the overhead associated with function calls. It does, however, increase the size of the resulting code since multiple copies of the function are made. Consequently, inlining is best done with small functions only. Although the programmer can declare that a function be compiled inline, whether it actually will be is determined by the compiler.

5.2 Object Oriented Language Features

We begin with a brief overview of a few object-oriented programming features to illustrate important features of the code which follows. We discuss these in the context of programming with C++, although they are shared by other object oriented languages. Special attention will be paid to the difference between procedural languages such as Fortran and C and the resulting differences in how code is designed.

Since this guide is far from exhaustive, readers interested in further details are encouraged to consult the references listed in the bibliography. The definitive guide is *The C++ Programming Language* by Bjarne Stroustrup [Str], who created the C++ language. This book contains a wealth of information on the language itself and the practice of object-oriented programming. Introductions can be found in [Oua] which is a general overview, and [Lad, Buz] which focus on applications to scientific computing.

5.2.1 Classes and Objects

A class is a module of code containing both data and functions which act on the data. This distinguishes it from the `struct` in C, which is a container for multiple pieces of data. (In C++, the `struct` is virtually identical to the `class`) An extensive

library of pre-defined classes is available for use with most C++ compilers. Additional classes are created by programmers to accomplish specific goals or to represent specific concepts.

Each class in C++ has a distinct *type*. These are distinct identifiers for each class, like the primitive data types which are built-in to the language: (`float, double, int, char` ...). This means that different distinct *instances* of the type can be defined inside a program, just as multiple integer variables are distinct instances of the `int` type. An instance of a class is called an *object*.

Data Members

The data which are part of a class are called its *data members*. These can be primitive types or objects of abstract classes. The functions which belong to a class are called its *member functions*. These functions accept arguments like other non-member functions, but can also access the data members of the class.

Inside an object, the member data and functions of an object can be accessed using only their name. Outside the object they are accessed through the notation, `object.DataMember` or `object.MemberFunc(`*`arguments`*`)` for a function. Where `object` is the name of the object, `DataMember` is the name of a data member and `MemberFunc` the name of a member function. We will see in the next section how to control which members can be accessed outside of the class they belong to.

When referring to members as part of a class and not as members of a particular object, we adopt the C++ syntax used in the class definition. This is `class::DataMember` or *`return-type`* `class::MemberFunc(`*`argument-types`*`)`. When the class name or the return type is clear from context, they can be omitted from the definition.

For more information on the syntax of defining classes and accessing members, see [Str, Chapter 10] and [Oua, Chapter 13].

Access Control

When the various parts of a class are defined, they are assigned a level of protection which determines where and how they can be accessed by other code. Both data and function members of a class can be declared *public*, which allows access both inside and outside the object they belong to. Members which are declared *private* can only be accessed by the member functions of that class or by special non-member functions which are declared to be *friends* of the class. (We will not examine friend functions in detail here; For more information see, [Str, p278].) A private data member can be accessed inside a member function of the class (and by *friend* functions), but nowhere else. A private member function can only be called by other member functions of the class (or by friends of the class). A third level of access, *protected*, lies between private and public and is relevant to classes related through inheritance, discussed in section (5.2.3).

A key part of object-oriented design is the use of private data members to selectively hide data from the other parts of the code. This is known as data encapsulation and its usefulness will be discussed below in the section on object-oriented programming.

5.2.2 Data Encapsulation and Interfaces

The practice of making data private inside an object is known as *data encapsulation* (or *data hiding*) and is used to limit the ways the contents of an object can be manipulated. For example, data which is necessary for the functioning of the object but not of interest to the user is often made private inside a class, where it is available to member functions that need it. The class then has the responsibility of keeping this information accurate. This can be guaranteed, since only the class itself has access to this data. In a procedural programming language, where there is no way to encapsulate data in an object implementing functions, the user of the code is often responsible for keeping track of this data and passing it to procedures as another

argument.

Consider for example a class `Matrix` which contains a dynamically-sized array of double precision numbers for holding its contents, and a pair of integers representing its dimensions. For consistency, the product of the dimensions should be equal to the size of the array. If the user wishes to resize the matrix, both the appropriate size value and the size of the array must be changed together. This is enforced by making the data and the size information private members of the class and providing a `resize` member function which implements the desired action. Other member functions are then provided for access to the data.

Encapsulation can be used to design classes which are easy to use. Functions written to use the `Matrix` class do not need to have separate arguments for accepting the data and the dimensions of the matrix separately, since these are bound together in a single object. The function can then use the same interface as the user for getting the necessary information from the class.

The publicly accessible part of a class is called the class' *interface*. This is the part of the class which is subject to manipulation by its clients, and an important part of class design is creating an interface which is flexible and easy to use.

Interfaces are even more important when data encapsulation is used to restrict the ways an object can be used. For example, data which is made private because it should not be freely manipulated may be needed outside the class. In this situation, public member functions are provided whose job it is to return the values of parameters which are private inside a class. These member functions are the means by which all other code can learn information about the class.

For reasons discussed above, the size information was made private inside the `Matrix` class. Since code which uses matrices will need to know this information, the interface of the class includes the member function `int get_size(int)` which returns the values of each dimension, depending on the argument.

5.2.3 Inheritance

Through *inheritance* new classes are defined by building on an already existing class. This new class is said to be *derived* from its *base* class. This is useful when the new class represents a specialization of the idea represented in the existing class. The derived class then extends or replaces the functionality of the base class as necessary.

For example, a vector can be thought of as a special kind of matrix where at least one of the two dimensions is equal to one, and whose elements can be referenced with a single index. To implement this relationship, a `Vector` class could be derived from our already existing `Matrix` class. The requirement that a dimension be equal to one would be enforced within the `Vector` class which would also implement accessing data through a single index. The remaining functionality and internal code would be inherited from the matrix class. The Matrix/Vector library described in Section 6.1 uses this implementation and we will expand further on this example in this chapter.

Inheritance is useful because it reduces code duplication. It is also one of two ways to create generic functions. C++ supports treating an object of a derived class as an object of its parent class under certain circumstances. For example, a function written to work with class `Matrix` could also accept a `Vector` object as its argument. This prevents the duplication of even more code by re-using the matrix version of the function. Note that this is different from *operator overloading* where different versions of a function with the same name exist for different argument types. In the case of a derived class, the object is literally "sliced" back to a base class representation and the same function is called.

5.3 C++ Terminology and Syntax

To clarify the following discussions, we define more C++ terminology and give some examples of correct syntax.

5.3.1 Syntax

C++ is a free-form language, which means that the programmer has great flexibility in the exact layout of the code and the layout is generally designed to enhance the readability of the code. A semi-colon (`;`) is used to terminate each statement and groups of statements are made with brackets {}. Unlike Fortran77, commands which are declarations of variables do not need to appear before the executable statements. They are allowed anywhere within any block of code.

Each program in C++ must contain one function called `main` which contains the main body of the code. The return type and argument types of `main` can depend on the compiler, although the arguments `main(int argc, char **argv)` are traditionally used for the processing of command line arguments which are passed to main.

5.3.2 Declarations and Definitions

To create large programs is it often convenient to separate the code into multiple files which store smaller parts of the whole. Since a compiler works only on one file at a time, there must be a way for the compiler to know essential information about classes and functions which have been, or will be defined in other files. To do this, C++ requires that classes and functions have a *declaration* and a *definition*.

The definition of a function or class is its complete specification. For functions this is the function's name, its return type, the names and types of its arguments and the statements which make up the function enclosed in curly brackets: { }. A declaration of the same function includes just enough information so that the compiler can check to make sure it is being called properly by other code. This requires just the name, return type and types of the arguments. The argument names do not need to be specified. This information about a function is also known as its *prototype*.

For example, `double square(double);` is a declaration of a function called `square` which accepts a single double precision number as its argument and returns

a double precision result. The complete definition might look like:

```
double square(double arg){
  return arg*arg;
}
```

A function must be defined exactly once in a program, and declared in every file in which it is used. These responsibilities are often split up into different files. The declaration of a function or group of functions is stored in a *header file*. On UNIX systems, these often end in the extension `.hh`. The definitions of these functions are stored in another file with the same name and the extension `.cc`. If a file contains a single function or class, it is customary to make the name of the file the same as the function or class it contains.

Each file of code which uses a function must contain the declaration of that function so the compiler can verify that it is used correctly. This is easily accomplished through the pre-compiler directive `#include "filename"` which inserts the contents of `filename` at the point where `#include` appears. For example, `#include "square.hh"` would include the header file of the square function. Pre-complier directives are processed automatically before the compiler begins work on the code.

Classes are declared and defined in much the same way functions are. The declaration contains the name of the class, all of its data members and declarations for all of its member functions. The full definition of the class builds on the declaration by providing definitions for each of the member functions. For this reason, the `.cc` file containing the class definition begins with `#include` to include its own declaration. Short member functions can also be given full definitions inside the declaration of the function. (On many compilers, this results in the member function being compiled as an *inline* function.)

For example, a class might be declared:

```
class widget {

private:
```

```
  int part_number;

public:
  int get_number(void);
  void set_number(int);
}
```

Class widget contains a single integer which is private and has two member functions which are public. This is just a declaration of the class because the member functions are only declared and not defined. The above would likely be stored in a header file called `widget.hh`. The corresponding `.cc` file might contain:

```
#include "widget.hh"
int widget::get_number(void){return part_number;}
void widget::set_number(int n){part_number = n;}
```

Note how much of the class information is included from the header file, so it is not repeated. Definitions are supplied for the member functions, completing the definition of the class. The file `widget.cc` could be compiled and linked into other programs containing widgets. This other code should also include the header file `widget.hh`.

5.3.3 Using Function and Class and Libraries

The `.cc` files containing the definitions of functions and classes can be compiled before becoming part of a program which used them, thereby saving time when the program is compiled. This produces an intermediate file, usually with an `.o` extension called an *object file* containing the pre-compiled code. The process of combining `.o` files into a program is called *linking* and is generally performed by the compiler as it compiles the main bode of the program. To use one of these pre-compiled files, the compiler is instructed to link the `.o` when compiling the program. As discussed before, the other files of code which use the functions or classes defined in the file must contain its prototype.

Libraries are simply related groups of object files combined into a single file to make the linking process simpler. With a library it is no longer necessary to know

the names of the individual object files in order to link their contents, When a library is linked with a program, it is searched for the components that the programs needs. On Unix systems, libraries generally have names in the form `lib`*`name`*`.a` and can be linked into a program with the compiler command `-l`*`name`*. Libraries often have their own header files which `#include` all of the header files of the components within it.

5.3.4 Essential Member Functions

A class created by the programmer can be endowed with any member functions desired. There are also three special member functions which must be defined in order for the compiler to properly handle objects of the class. These member functions are *constructors*, *copy constructors* and *destructors*. These member functions are important enough that if the programmer does not define them for a new class, the compiler takes it upon itself to define them.

Constructors

The creation of an object is known as its *construction* and is performed by calling a *constructor* member function of the class. Special actions necessary upon creating the object are placed inside a constructor member function. For example, whatever dynamically allocated memory an object of the might use can be allocated in a constructor. The constructors of a class all have the same name as the class itself and they return no arguments. There can be more than one constructor for a class, so long as they accept different arguments.

The arguments of a constructor can be used to determine the properties of the resulting object. For example, the `Matrix` class has a constructor `Matrix::Matrix(int, int)` where the two integer argument are the dimensions of the resulting matrix. Creating an object of the `Matrix` class discussed looks like:

```
Matrix M(5,4);
```

Here, `M` is the name of the `Matrix` object being created and `(5,4)` are arguments being passed to the constructor. The resulting `Matrix` object would represent a 5 by 4 matrix.

Copy Constructor

This is a special constructor which creates a new object from an existing one. This is done automatically by the compiler when an object is returned from a function, or passing an object as an argument to a function. In both these cases a temporary copy of the object is created using the copy constructor of the class. Like all constructors, they have the same name as the class and are distinguished by their argument. The exact specification of a copy constructor is: `ClassName::ClassName(const &ClassName)`. This indicates that the function's argument is a reference to an object of the same class, which it treats as a constant. If a copy constructor is not defined for a class, the compiler creates one which invokes the copy operators of each of its data members. A copy operator should be provided by the programmer whenever this is not the desired behavior, as is often the case in classes which use dynamically allocated memory.

Destructors

The *duration* of an object is the extent of the code during which it exists and other code can use the object. When an object's duration ends, a special member function called the destructor is called. For classes which dynamically allocate memory, the destructor is where the de-allocation of memory is usually performed. Only one destructor can be defined for a class. It takes no arguments and has the special name `ClassName()`. If no destructor is defined for a class, the compiler creates one which simply invokes the destructors of the class' data members.

5.3.5 Operators for Classes

The *operators* of a language are the symbols which cause certain operations to be performed on objects when invoked in the code. For example the common arithmetical operators (`+,-,/,*`) are defined to operate on all of the primitive numerical types in C++. When a programmer creates a new class, the behavior of these operators on objects of the class can also be defined. This is usually done by defining special member functions of the class which correspond to these operators. The operators we are most concerned with are parentheses and the *left-shift operator* `<<`.

To use a parenthesis operators with objects of a class, the class must define the special member function *`return-type`* `ClassName::operator()(`*`argument-types`*`)` and the operator is then invoked by `ObjectName(`*`arguments`*`)`. This is equivalent to the traditional notation for member functions: `object.operator()(`*`arguments`*`)`. For example, in class `Matrix` the parenthesis operator is defined to allow access to individual elements with notation like `M(i,j)`. The prototype of the member function which does this is `double& Matrix::operator()(int, int)`, which takes two integer arguments and returns a reference to a double value.

The left-shift operator is usually designed to work with C++ classes called *streams* for outputting the contents of a class to the screen or a file. The most commonly used output stream has the pre-defined name `cout` which sends information to the standard output. Its usage with the left-shift operator looks like: `cout << object`. A left-shift operator designed to work like this would have the prototype `ostream& operator<<(ostream&, Class &)`. The class `ostream` is one type of C++'s output stream classes.

5.4 Generic Programming

Generic programming means the design and implementation of code which can operate on a variety of different types and implement the correct functionality for each. It

is another code paradigm distinct from, but supported by, object-oriented programming.

Object-oriented design often results in the creation of many new classes, each represented by a different type. Situations arise when it is desirable to apply the same code to these different types. Furthermore, C++ is a strongly typed language, which means that the compiler will check to see that the objects passed to a function are an exact match, or acceptable substitute, for the types in its definition. To avoid re-writing functions to work with every different class, there is a need to write functions which work with a variety of object types.

There two ways to write functions in C++ which can act generically over different types of objects. Which method should be used depends on the kind of generic behavior desired. The two types of generic code are *run-time* and *compile-time* and depend on when the type of the argument is known.

5.4.1 Polymorphism and Run-Time Genericity

If the goal is to write functions which will accept arguments whose exact type is not known *until the program is actually being run,* then run-time genericity is desired. There are significant performance issues surrounding the use of run-time genericity which prevent us from applying it in this project.

The basic idea behind run-time genericity is called *polymorphism* which means that the responsibility of providing the correct functionality belongs to a class which implements it (usually as a member function). Varying functionality is attained at run time by using objects of these different types. The mechanism which makes this possible has already been encountered in Section (5.2.3) on inheritance. The different classes which are to be used by a function are derived from a common base class. The function which needs to behave polymorphically is then written to work with the base class as its type. Objects of the various derived types are then when the program executes. The key here is that the compiler allows derived types to be passed to a function in the place of base types.

This kind of genericity requires that functions be given a special declaration to ensure that the functionality of the *derived* class is invoked and not the *base* class which actually appears in the definition. These are known as *virtual* functions and they require an extra layer of indirection to be executed. Because the compiler does not know the exact function, the function call is implemented through a function pointer, whose address is set when the code is run. This is partially responsible for the performance problem with these kind of functions. The rest of the difference is caused by the inability of the compiler to *inline* virtual functions, which means that the function itself is copied directly into the code where it is called. This isn't possible because the compiler simply doesn't know what the function is. For more information, see [Str, Chapter 12]

5.4.2 Templates and Compile-Time Genericity

Compile-time genericity refers to code written to act generically with different types which are known when the code is compiled. This is implemented through a C++ language construct called the *template*. With a template, functions and classes can be written with place-holders substituted for one or more unknown argument types. When the compiler finds a templated function being used, a version of the function is created and compiled where the correct argument types are substituted for the place-holders. When templated classes are created, the types to be substituted are given explicitly. These specific versions of a template are called *instantiations* of the template. Note that compile-time genericity does not require virtual functions and thus avoids the associated performance problems.

We illustrate the difference between the definition of a template and an ordinary function with an example.

```
template <class T>
T min(T a, T b){
  if (a<b) return a;
  else return b;
};
```

The statement `template <class T>` declare the following definition to be a template on the unknown type `T`. (Primitive data types can be used, even though `T` is called a class.) The template function `min` is then defined which takes two arguments of type `T` and returns one. When the function `min` is invoked in code, for example by: `min(4.5, -8.9)` a version of the `min` function is created with an appropriate data type. In this , `float`. Note that this function can be instantiated with any type for which the operator `<` is defined. This can include user-defined classes through the definition of `operator<`, as discussed in Section 5.3.5. Invoking `min` on a class which does not support this operator would result in a compiling error.

A templated class is defined as in the following example.

```
template <class Key, class Target>
class Pair {
  Key theKey;
  Target theTarget;
  ...
}
```

Here, there are two template place-holders, `Key` and `Target`. The resulting class is called `Pair<Key, Target>` and contains an object of `Key` type and an object of `Target` type. (This class template would be useful in the context of database programming for creating indexed lists of various types.) When objects of this type are created, the full name of the type must be specified:

```
  Pair<int, String> employee;
```

Here, `employee` is an object of type `Pair<int, String>`.

Templates are a recent addition to C++ and their support by various compilers is somewhat irregular. To date, many of the popular and freely available compilers do not fully support templates, so every language construct available in a non-templated class is not always available in a templated one. For example, the popular gcc compiler from the Gnu Free Software Foundation does not support static members or inlining of member functions in templated classes in version 2.7.2. Although we won't go into

these issues here, they have important ramifications for the flexibility and performance of the resulting code.

We've barely touched on the use of templates in this introduction. For more information see, [Str, Chapter 13], [Oua, Chapter 24], [Mur, Chapters 7,8] and [Lad, Chapter 1].

5.5 Using Objects to Represent Functions

The use of functions in code to represent mathematical functions suffers from a few key limitations. We discuss these here, and show how the use of objects to represent functions can solve these problems.

When a code function is written, it necessarily represents a single mathematical function. For example:

```
double theFunc(double x) {
  return 0.5*x*x + 0.25*x + 1;
}
```

implements the quadratic function $\frac{1}{2}x^2 + \frac{1}{4}x + 1$. If we want a more general quadratic function, we might write it as follows:

```
double A,B,C

double quadFunc(double x) {
  return A*x*x + B*x + C;
}
```

Note that the coefficients `A,B` and `C` are defined *outside* the function. This is necessary if they are to be accessible to other parts of the code. (In Fortran, this might be done with a `common` block.) Now we can implement various quadratic functions, but this code can only represent only one function at a time. There is no way to implement more than one quadratic function at a time without duplicating this function. The second version of the function and its coefficients would require different

names, because names can not be duplicated within the same scope. This would complicate writing other code which works with these functions.

In situations where only one instance of a particular function is needed and will be used in a limited variety of ways, then this method can be made to work. This is more common for functions which are being used for the high-level organization of code, rather than as mathematical functions.

For mathematical functions, and functions which can be used in a variety of settings, this approach is inadequate. We need a way to create functions which can be manifested as different instances whose data is independent. This is accomplished by writing a class.

Consider the following class.

```
class Quad {
public:
   double A, B, C;
   double operator()(double x) { return A*x*x + B*x + C; }
}
```

This defines a class which represents quadratic functions. It can be used as follows:

```
void main(){
   double result;
   Quad func1;
   func1.A = 0.5;
   func1.B = 0.25;
   func1.C = 1;
   result =  func1(1.0);
   Quad func2;
}
```

The second through fifth statements create a `Quad` object called `func1` and assign the value of its three coefficients. The sixth statement evaluates the quadratic function represented by `func1` and assigns it to `result`. This works by calling the member function `operator()` which implements the function-like calling method used here. The last line creates another `Quad` object whose coefficients are completely independent of those in `func1`.

Chapter 6

Applications of Object-Oriented Design

The goals outlined in the introduction to Chapter 5 have lead to the development of three bodies of code representing different levels of generality in the computations they perform. The less general code is more specifically tailored to the computational problems discussed in Chapter 4 and uses the more general code to handle its basic operations. We will describe these bodies of code beginning from the most general elements and finishing with the most specific.

The first level consists of code which performs fundamental mathematical operations. It includes a library of classes representing matrices and vectors and code which implement some of the familiar operators of numerical linear algebra. This level also contains operators for numerical quadrature, integration of systems of ordinary differential equations and optimization of functions.

Many parts of this library, perhaps especially the first level, are narrower in application than would be expected of a general purpose library. For example, the Matrix/Vector library only implements rectangular matrices and square banded matrices. Factorization of these types of matrices is limited to LU decompositions. This is due to the purpose for which the library was developed.

Features were added to the library as they were deemed useful for solving the particular problems of interest in this research project. Efforts were taken, however, to write code which would support the addition of new features and the eventual

expansion into a complete library. This is done in various parts of the library by isolating the details of algorithms in classes which represent them. This class can then be replaced by others playing the same role, but implementing different methods. We will encounter examples of this design approach in all three levels of code.

The second level involves the creation of classes for use in one-dimensional Galerkin finite element calculations. It includes classes which represent one dimensional grids defined on an interval, and classes which represent the shape functions from which basis functions are assembled. These classes are used in turn by other classes which represent a set of Galerkin basis functions on a particular grid. These classes are somewhat less general than those in the basic library and their functioning obeys specific inter-relationships which are more complex. The dependence relationship between classes has been kept one-way whenever possible to simplify the development and testing of code.

The third level of code is the most specific and contains classes designed to solve the particular finite element problems from Sections 4.5,4.3 and 4.7 of this paper. The inter-relationships between the various classes are even more exacting, preventing their use in a really generic way. However, even at this level of code attention has been paid to a design which separates the various parts of the problem, and allows for different problems to be solved within the same framework.

6.1 The Matrix/Vector library

6.1.1 Design Goals

The Matrix/Vector library was designed to support the following features.

- Dynamic sizing.
- Responsible memory management.
- Size checking.
- Minimizing memory allocation and data copying.

- Reuse of Fortran LAPACK.

We examine each of these items in more detail.

Dynamic Sizing

Matrices and vectors can be created with sizes that are not determined until the code is being executed and are not limited by constraints determined when the code is compiled. Furthermore, a matrix or vector can be resized during the execution of the program. (Note that this is a kind of run-time genericity which depends on C++'s ability to dynamically allocate memory.) This overcomes a serious limitation of Fortran 77, in which matrices are stored as part of arrays of memory which have fixed sizes. The size of the array limits the size of the matrices it can contain and cannot be changed without re-compiling the code.

The dynamic sizing feature is usually used to allow the size of objects to be specified when the program is run, rather than resize an already created object. Because it involves interaction with the operating system, allocating and deallocating memory is a slow operation, so resizing objects is avoided in the rest of the library or applications which use it. Furthermore, resizing a matrix or vector object always discards its contents.

Memory Management

A consequence of using dynamically allocated memory is the need to de-allocate memory which is no longer needed. (See Section 5.1.4) Objects in the Matrix/Vector library are therefore designed to keep track of and de-allocate memory as needed.

Size Checking

Matrix/Vector or Matrix/Matrix operations are not allowed on objects which are not of compatible sizes. The mechanism for checking the sizes is designed to not excessively burden the execution of the code.

Minimizing Memory Allocation and Data Copying

The process of allocating memory and copying data are slow, so efforts are taken to minimize the number of times this happens in the course of a program. The solution we implement is to allow a matrix or vector to act as a "view" of data belonging to another object. These views can be used to pick out subsets of the data contained in the other object without a copy operation. This has implications in the memory management part of the code, since an object acting as a view into data it does not own is not responsible for de-allocation of the data.

Reuse of Fortran LAPACK.

All three levels of the Basic Linear Algebra Subprograms are accessible through functions which apply them to the matrix and vector objects in the library. This is done by taking the necessary data from the matrix or vector object and calling the compiled versions of the Fortran BLAS routines. Similarly, LU factorizations are implemented for the matrix classes by calling the appropriate LAPACK routines. LAPACK is used for these jobs because it is a widely used, freely available library which exists in highly optimized forms for various computers. The NIST project [LAPACK++] is another example of this approach.

This design decision has wide reaching implications for how the library is implemented. For example, data for matrices is stored by column in a single one dimensional array, because this is the way Fortran arrays are stored. Likewise, the data for banded matrices is stored in the banded storage format used by various LAPACK routines.

6.1.2 Interface of the Matrix/Vector Library Objects

We begin by describing the external interface that the various classes in the library present to users. This is the key information for using the library within an application. The material presented here is an overview. A detailed summary is in

Appendix A

The library consists of three elementary types: `Matrix, Vector, BandedMatrix` representing matrices, vectors and banded matrices respectively. Two derivative types: `MatrixLU, BandedMatrixLU` implement LU factorizations and solve linear systems for the matrix types. Internally `BandedMatrix` uses a banded storage scheme. Externally, it functions much the same as `Matrix`.

Owner/View operation

The `Matrix` and `Vector` classes have two distinct modes of operation. They can either own their data, or act as views into data which is contained in another object or by other code. Functions for accessing and changing data, and all functions which operate on `Matrix` and `Vector` objects function identically whether the object is an owner or a view. At this time, there is no view support for the `BandedMatrix` class because of the internal storage requirements.

Construction

Recall that objects are created in a C++ program through the process of construction. Each class has various constructor member functions to which various arguments are passed at construction.

The three elementary classes can all be constructed with no arguments, resulting in matrices and vectors of size zero, or with integer arguments that specify their size. Objects of class `Vector` require a single argument for the size, and can optionally accept a second boolean argument indicating whether the object should be a column or row vector. (Column vectors are the default and are indicated by the boolean value `true`.) `Matrices` are constructed with two arguments giving the number of rows and columns. Since the `BandedMatrix` class only supports square matrices, these objects are constructed with three parameters giving the size, the number of upper bands and the number of lower bands. Most operations on vectors, matrices and banded matrices of size zero are illegal, except those which resize the object.

The `LU` classes are usually constructed with a `Matrix` or `BandedMatrix` argument. The data of the construction argument is copied and factored inside the `LU` class object, leaving the original version intact. An object of these classes can also be assigned to another matrix or banded matrix, using the `copy` member function with a `Matrix` or `BandedMatrix` argument. This is necessary if no matrix was specified at construction.

The values in a matrix or vector are not initialized by the library code when it is constructed. Some compilers will automatically set the contents of allocated memory to zero while others do not, leaving them with contents that are essentially random. When using such a compiler, it is important to assign a value to every element in the object. As seen in the section on element reference, an entire matrix, vector, or banded matrix can be set to zero with one line of code: *`object`*`=0`

Resizing

The `Matrix`, `Vector` and `BandedMatrix` classes can be resized after construction by calling the `resize` member function and specifying a new size. This is required when using an object constructed with a zero size. Objects can also be automatically resized to match the size of another object of the same type with the `resize` command.

Initial Data

`Matrix` and `Vector` classes can be assigned initial data upon construction through the constructors `Matrix::Matrix(double*, int, int)` and `Vector::Vector(double*, int)`. If the pointer argument is non-constant, the object becomes a view of the data, if the argument is constant, a copy is made and the object owns the data. Objects of class `BandedMatrix` cannot be constructed this way.

Element reference

All three elementary classes support the parenthesis notation for referring to individual elements. All indices begin at 0. For example, `M(2,4)` is the matrix element in the third row and fifth column of `M` and `v(2)` is the third element of vector `v`. This can also be used for assigning values to the elements, i.e., `M(2,4)=10.0`. This operator also works for banded matrices and automatically accounts for the banded storage scheme. If the element referred to is outside the bands, 0 is returned as its value and assigning to it has no effect. These classes also support assignment to all elements by a scalar value using the operator `=`.

Output

Output of objects in the library is handled through C++'s `stream` classes. These are a large library of classes which support formatted and unformatted output to standard devices including files. Implementation is done though the left-shift operator: `operator<<(ostream& B)`

Examples:

Consider the following fragment of C++ code:

```
#include"Matrix.hh"
#include"Vector.hh"
#include"BandedMatrix.hh"
#include"blas++.hh"

void main(void){

  Matrix M(6,6);
  Vector x(6), b1(6), b2;
  BandedMatrix B;

  B.resize(6,1,2);
  b2.resize(b1);
```

```
  for (int i=0; i<6 ; ++i) {
    for (int j=0; j<6 ; ++j) {
      M(i,j)=1.0/(1+i+j);
    }
    x(i) = i;
  }

  B = 2.0;
  B(0,5)=100;
  for (int i=0; i<6 ; ++i) B(i,i)=-1.0;

  cout << "M:" << endl << M << endl << endl;
  cout << "B:" << endl << B << endl << endl;
  cout << "x:" << endl << x << endl << endl;

}
```

The first four lines are directives to include the appropriate header files as described in Section 5.3.3. The first three contain class prototypes and the last is a library header file for the BLAS routines.

The next line begins the `main` function which every program must contain. This is where the execution of the program begins. The matrix `M` and vectors `x` and `b1` are constructed with non-zero sizes, while the banded Matrix `B` is constructed with size zero, and resized to 6 by 6 with one upper and two lower diagonals. The vector `b2` is constructed with zero size and resized to match the size of `b1`.

The next statements are nested `for` loops which set the value of the elements in `M` and `x`. Matrix `M` becomes the Hilbert matrix of size 6, where $M_{ij} = \frac{1}{1+i+j}$ and $x_i = i$, for $i, j = 0, \ldots, 5$.

The next three lines set the values of `B`. First, all of the elements in the bandwidth of B are set to 2.0. This operation does not change the value of elements outside the bands. The next line is an attempt to assign to an element of `B` which is outside the bands and which has no effect. Then all of the diagonal elements are set to -1 inside another for loop.

The statements beginning with `cout` are output statements. They output a label and the contents of `A, B` and `x`, respectively. (`endl` is used to terminate each line of output.) The output is:

```
M:
1 0.5 0.333333 0.25 0.2 0.166667
0.5 0.333333 0.25 0.2 0.166667 0.142857
0.333333 0.25 0.2 0.166667 0.142857 0.125
0.25 0.2 0.166667 0.142857 0.125 0.111111
0.2 0.166667 0.142857 0.125 0.111111 0.1
0.166667 0.142857 0.125 0.111111 0.1 0.0909091


B:
-1 2 0 0 0 0
2 -1 2 0 0 0
2 2 -1 2 0 0
0 2 2 -1 2 0
0 0 2 2 -1 2
0 0 0 2 2 -1


x:
0
1
2
3
4
5
```

Notice that the off-diagonal elements of `B` are still zero.

A superset of the BLAS routines are implemented for the `Matrix`, `Vector` and `BandedMatrix` classes. Most of the level one BLAS are written so that they can be applied to matrices and banded matrices as well as vectors. Furthermore, the multiplication of banded matrices with regular matrices is supported.

Continuing the sample code above, the following code (inserted inside the `main` function) illustrates some of these functions.

```
  cout << "|x|_1: " << Blas_Norm1(x) << endl;
  cout << "|x|_2: " << Blas_Norm2(x) << endl << endl;

  Blas_Mat_Vec_Mult(M,x,b1,1.0,0.0);
  Blas_Mat_Vec_Mult(B,x,b2,1.0,0.0);

  cout << "b1:" << endl << b1 << endl << endl;
  cout << "b2:" << endl << b2 << endl << endl;
```

The first two lines compute and output the l_1 and l_2 norms of the vector `x`. Line four computes the matrix-vector product Mx and stores the result in the vector `b1`. Note this this function is built upon the LAPACK routine DGEMV, which implements the operation: $\alpha Mv + \beta x \rightarrow x$. The last two parameters in the function call are the values of α and β. The last line computes the product Bx and stores the result in `b2`. The final lines then output the contents of `b1` and `b2`. The result is as follows:

```
|x|_1: 15
|x|_2: 7.4162

b1:
3.55
2.81429
2.34643
2.01746
1.77183
1.58074


b2:
2
3
6
11
16
9
```

The full list of BLAS routines is given in appendix B.

Using Factorizations

The factorizations in objects of the `LU` classes are then used through the member function `solveIP(Matrix& B)`, which accepts a matrix or vector as an argument. The function computes the solution of the linear problem $MX = B$ where M is represented by the factorization object and B is given in the argument. The computation is done in place, so the result is returned in the matrix argument `B`.

Continuing the above example, we add the following statements to `main`.

```
MatrixLU MLU(M);
BandedMatrixLU BLU;

BLU.copy(B);

MLU.solveIP(b1);
BLU.solveIP(b2);

cout << "inv(M)b1:" << endl << b1 << endl;
cout << "inv(B)b2:" << endl << b2 << endl;

Blas_Add_Mult(b1,-1.0,x);
Blas_Add_Mult(b2,-1.0,x);

cout << "Error in matrix solve: " << Blas_Norm2(b1) << endl;
cout << "Error in banded matrix solve: " << Blas_Norm2(b2) << endl;
```

This creates a `MatrixLU` class which is assigned to matrix `M` upon construction, and a `BandedMatrixLU` class `BLU` which is originally unassigned. The next statement assigns `BLU` to the banded matrix `B`. The following two lines solve the problems $Mx = b1$ and $Bx = b2$, storing the results in vectors `b1` and `b2`. After outputting the two solutions, x is subtracted from each and the norm of the residual is computed.

```
inv(M)b1:
6.22113e-13
1
2
3
```

```
4
5

inv(B)b2:
3.88578e-16
1
2
3
4
5

Error in matrix solve: 3.78112e-10
Error in banded matrix solve: 1.57499e-15
```

6.1.3 Implementation of the Matrix/Vector Library

All of the objects in the Matrix/Vector library use double precision real numbers as their basic type. We briefly summarize the implementation of each class in the library with emphasis on its data members and the part of its interface responsible for accessing the data.

doubleVector

The class `doubleVector` was created to handle the dynamically allocated arrays of double precision numbers used in all of the other classes in the library. It implements access to the data through `operator[]`, prevents access to elements which are out of range, and assumes the responsibility of deallocating the data when its lifetime expires. It also keeps track of whether it owns the data or is acting as a "view". Objects of class `doubleVector` also return a pointer to the data it stores through the member function `double* addr(void)`. This is for use with the Fortran LAPACK routines, which expect pointer arguments for arrays of data.

Matrix

Class `Matrix` has a data member object of type `doubleVector` which contains its data. Class `Matrix` is also, directly or indirectly a base class for every other class in the library, thereby providing the data storage for each of these classes. It also stores the two dimensions of the matrix as integer data members and implements access to its data through the parenthesis notation discussed in Section 5.3.5. This is done by mapping the row and column indices to the correct element in its `doubleVector` data member.

BandedMatrix

Class `BandedMatrix` is inherited from `Matrix` and so the `Matrix` parent provides the data storage and part of the data accessing interface. Class `BandedMatrix` is responsible for implementing the remaining part of the data accessing routine and storing the number of upper and lower bands in the matrix.

Objects of this class use the same banded storage scheme as the factorization routines in LAPACK. In this scheme, a square banded matrix of size N with l lower diagonals and u upper diagonals is stored as a matrix with $N+1$ columns and $N+2l+u$ rows. The bands of the original matrix become rows l through $N + 2l + u - 1$ of the storage matrix while the columns of the original matrix are preserved. (The extra column is not required by LAPACK, but is included to prevent accidental out of bound accesses by other routines.) Rows 0 through $l-1$ of the storage matrix are not used for storage, but are included for use by LAPACK factorization routines. The mapping from logical coordinates (i, j) to the storage coordinates $(\hat{\imath}, \hat{\jmath})$ is

$$\begin{aligned} \hat{\imath} &= l + u + i - j \qquad \text{provided} \quad -u \leq i - j \leq l \\ \hat{\jmath} &= j \end{aligned}$$

`BandedMatrix` implements this mapping between the logical and stored matrices. Then the `Matrix` parent handles the mapping of the stored matrix coordinates into the data held by the `doubleVector`.

Vector

We view vectors as matrices which have (at least) one dimension of size one. Consequently, we derive class `Vector` from class `Matrix`. Additional code in class `Vector` enforces the requirement that one of the dimensions be equal to one. This class also implements single index access to the data through the operator `operator()(int i)` by calling either `Matrix::operator()(i,0)` or `Matrix::operator()(0,i)` as appropriate.

LU Factorization Classes

The classes `MatrixLU` and `BandedMatrixLU` are both inherited from `Matrix` and `BandedMatrix` but not in the same manner as discussed before. A more restrictive means of inheritance called *private inheritance* is used. This is different from the public inheritance discussed earlier in that the components of the base class are accessible only *inside* the derived class, even if they are public in the base class. In effect, the base class is a private part of the derived class. We use this type of inheritance here because the behavior of an `LU` class is sufficiently different from that of its base class that we do not want to be able to treat is as one. Recall that when classes are inherited publicly, C++ supports treating an object of the derived class as if it were an object of the base class in some situations.

The `LU` classes also require that the matrix object they represent be square. When constructed with or reassigned to a `Matrix` object, the copy operator of the parent class is invoked with the target matrix as its argument. This creates a copy of the target accessible only inside the `LU` object. The `LU` class then calls the appropriate LAPACK routine (DGETRF for `Matrix` and DGBTRF for `BandedMatrix`) for factoring the matrix type and stores the pivot information generated by these routines. These classes also solve linear systems using the factorizations they represent. The `solveIP` member function takes a `Vector` or `Matrix` argument and passes it to the correct LAPACK routine (DGETRS or DGBTRS) with the factorization and pivot

information. The result is returned in the argument.

6.2 Function Classes

A small number of classes have been written to implement particular functions. The creation of new classes to represent different functions is a simple matter. All of these classes implement function evaluation through the member function `operator()`. We summarize the most important of these classes here. These are detailed further in Appendix C.1

6.2.1 Class `Polynomial`

The first of these is class `Polynomial` whose objects represent polynomials of arbitrary degree. This class supports the assignment of individual coefficients through the bracket notation `[]`, i.e., if `poly` is an object of class `Polynomial` then `poly[0]` represents the highest-order coefficient. Polynomials are constructed with a degree and can be resized through the member function `set_degree(int)`. The number of coefficients is one more then the degree of the polynomial, so the coefficients are `poly[0]` through `poly[degree]`. The value of the degree can be obtained through `get_degree()` and the polynomial is evaluated through `poly(`*`value`*`)`. The primary use of class `Polynomial` is in class `ShapeFunctionsC0`, which uses them to represent the shape functions from which the basis elements of a Galerkin approximation scheme are made. This is described further in Section 6.3.2.

6.2.2 Class `Harmonic`

Class `Harmonic` is used to represent the function $A\sin(2\pi\times f+\phi)$, where the amplitude A, the frequency f and the phase shift ϕ are all specified at construction.

6.2.3 Class `Indicator`

This class represents the indicator function on an interval. The upper and lower bounds are specified at construction time and the function evaluates to 1 or 0 depending on whether or not the argument is inside the interval. The interval is considered to be closed, so the function evaluates to one at each of its end-points.

6.2.4 Class `Windowed`

This is a template class which applies an indicator function to an arbitrary function of another type. It is constructed by providing this other function and either a range of values for the interval or an object of class `Indicator`. This class and class `Harmonic` are used to represent the windowed interrogation signals used throughout the simulations. The actual class in use is `Windowed<Harmonic>`.

6.3 Finite Element Classes

The next level of code consists of classes and functions useful in the calculation of one-dimensional finite element problems. These were developed with their usefulness to the problems from Chapter 4 in mind, but have applications to other problems as well. The further development of this level of the code would be of great benefit to other scientific applications.

Class Families

In this section and the next, it will be useful to discuss groups of classes called *families* whose members serve similar roles in different applications. The classes in a family may be related to one another through inheritance or by being different instantiations of a templated class. A family is often defined by the requirement that its members implement a common interface. This would mean that all of the classes in a family would all implement a common set of public member functions and have a common

set of public data members. One reason for so defining a family of classes is for use as arguments of a templated class or function which uses that particular interface.

6.3.1 Grid Classes

Members of the family of grid classes represent a partition of an interval into a finite number of points. Classes in the family must implement the operator `operator()(int n)` which returns the coordinate of node n, and the member function `d(int n)` which returns the distance between nodes n and $n+1$, and `size(void)` which returns the number of nodes.

There are currently two classes in the grid family. The class `grid` is the more general one, allowing for any kind of distribution of its nodes. Class `uniformGrid` is inherited from `grid` and represents uniform classes only. All of the basic functionality required by classes in this family is inherited from the base class `grid`. Following this pattern of development makes the addition of specialized classes very simple, since then can inherit the minimum necessary functioning from this class. The complete interfaces of these classes are given in Appendix E.1.

6.3.2 Galerkin Finite Element Classes and Functions

The main class in this family is the template class `GalerkinC0<G>`, whose template argument specifies the type of the grid class on which it is based. Internally it contains a `BasisFunctionsC0` class, which in turn uses the `ShapeFunctionsC0` class.

ShapeFunctionsC0

The `ShapeFunctionsC0` class represents information about shape functions of a certain degree defined on the reference interval $[-1, 1]$. These are the building blocks from which the Galerkin basis functions are built. The desired degree is the only information specified upon construction of the object. The only supported values for the degree are $d = 1, 2, 3$.

Once created, the `ShapeFunctionsC0` object provides information about the shape functions. It supports the evaluation of the shape functions $s_i(\xi)$ through `double ShapeFunctionsC0::operator()(int i, double z)`, and the shape function derivatives $s_i'(\xi)$ through `double ShapeFunctionsC0::deriv(int i, double z)`. Values of the following matrices are provided:

$$\begin{aligned} M_{i,j} &= \int_{-1}^{1} s_i(\xi) s_j(\xi) d\xi \\ W_{i,j} &= \int_{-1}^{1} s_i'(\xi) s_j(\xi) d\xi \\ K_{i,j} &= \int_{-1}^{1} s_i'(\xi) s_j'(\xi) d\xi \end{aligned}$$

Individual elements are referenced through member functions `M(int i, int j)`, `W (int i, int j)` and `K(int i, int j)`, or the entire matrices can be retrieved as pointers to double with the member functions `Mmatrix(void)`, `Wmatrix(void)`, and `Kmatrix(void)`. The complete interface is detailed in Appendix E.2

Basis Functions

Class `BasisFunctionsC0` uses an instance of class `ShapeFunctionsC0` internally. It represents the basis functions on an interval but does not contain the geometric information. without reference to any particular domain. Instead, it is concerned with the composition of the basis functions from shape functions, the relationship of the shape functions to the intervals of the domain, and whether or not these functions are zero on the boundaries. Its interface is described fully in Appendix E.3. Its primary role is to assist in the computations of a `GalerkinC0<G>` object.

Galerkin Objects

Class `GalerkinC0<G>` contains a grid object of type `G` and a `ShapeFunctionsC0` object. The result is a representation of the basis functions $\phi_i(x)$ on the interval specified by the grid object. They can be used to evaluate the basis functions at points in the

domain and compute parts of the mass, stiffness and flex matrices. The entire interface is covered in Appendix E.4

Most currently existing applications use the `GalerkinC0<uniformGrid>` instantiation of the `GalerkinC0<G>` template. For the numerical problems of Section 4.5 two such classes are used to represent discretizations of the space (z) and history (s) variables.

Various functions are written to operate on `GalerkinC0` classes. These perform operations such as computing the vector of inner products $\langle \phi_i, f \rangle$ and the matrix $\langle \phi_i f, \phi_j \rangle$ where the ϕ_i are the basis functions and f is an arbitrary function. These functions are described in Appendix E.5

6.4 Differential Equation Classes and Integration Classes

First-order differential equations of the form $\dot{y} = f(t, y)$ are represented in this library as classes which implement the operation of computing $f(t, y)$. Classes in this family must implement two member functions: `void operator()(double, Vector, Vector)` and `int get_size()`.

The member function `void operator()(double, Vector, Vector)` is for evaluating the derivative. The first two arguments are the time t and state y. The final vector argument contains the derivative $\dot{y}$ after the function is called. The member function `int get_size()` returns the size of the system of differential equations.

We examine briefly an alternative design for the family of differential equations classes which was tested and rejected for reasons of efficiency. In this version, evaluating the derivative is done through the member function `Vector operator()(Vector, double)`. Here the derivative is returned through the function return mechanism. This syntax more closely resembles mathematical usage of the functions, since it allows statements like `deriv = equation(time,state)`.

This approach is considerably less efficient than the one described above because

of how C++ handles objects returned from functions. Since C++ normally passes arguments (and return arguments) by value and not by reference, a temporary copy of the object to be returned is created (using the default copy operator of the class). For `Vector` objects, this means allocating more memory and copying the results into it. Outside the function, another copy operation is required from the temporary object into the left-hand-side vector. In the example above, the data in the temporary object returned by `equation(time,state)` must be copied into the vector `deriv`. Then the temporary object is destroyed and its memory de-allocated. Every function evaluation thus requires two extra memory operations and two extra copy operations. These operations are avoided by providing the storage space for the result as another argument to the function.

6.4.1 Integrators and System Classes

The library currently contains two methods for integrating systems of ordinary differential equations. The first is written entirely in C++ and implements a fourth order Runga-Kutta method. The second is an interface to the `odepack` routine `lsoda`, an adaptive stiff/non-stiff solver.

An Integrator is expected to support the following member functions in its interface:

```
void set_time(double)
double get_time(void)
void set_state(Vector)
Vector& get_state(void)

void integrate_to(double)
void advance(double)
```

The meanings are explained further in Appendix D.1, but most are clear from the function name. Note that `get_state` returns a reference to a vector to avoid a copy

operation. Hence, this function can also be used to modify the state of the system. Additional integrators added to the library may implement other functions as well; These are the bare minimum that any integrator should implement.

In order to work with a variety of functions, integrator classes are templated over the type of the function they are applied to. These functions are expected to belong to the differential equations family, i.e., they must implement the member functions `void operator()(double, Vector, Vector)` and `int get_size(void)`.

The integrator class keeps track of the the current time and state of the system. In effect, it represents the entire differential equation including the function and the state of the system, as well as implementing a numerical approximation scheme. For this reason, we also refer to these as *System classes*, especially in the context of discussing the forward and inverse problem classes of Section 6.8.

Class `rkMethodIP<F>` implements a fixed step size fourth order Runga-Kutta method. Its complete interface is given in Appendix D.1.1.

The `lsoda` routine is accessed through a different syntax which is designed to accommodate more routines as the library grows. The lsoda routine itself is represented by a special class called `lsodaMethod` and the routine is combined with a differential equation function through the templated class `SystemIntegrator<F,M>`. Here the template argument `F` is the type of the differential equation class and `M` is the type of the method class. The choice of method class is currently limited to `lsodaMethod`. The `SystemIntegrator` class implements the system class interface, providing a common interface to various methods represented by their own classes.

6.5 Optimization Routines

The class `Optimizer` acts as an interface to a Fortran optimization routine that was originally chosen for the implementation of the Debye Model inverse problem in Section 4.2. The class is designed to work with objective functions of the form

`void funcName(int*, double*, double*)` which represents a mathematical mapping $\mathbb{R}^n \rightarrow \mathbb{R}$. Note that there is no return argument, and all of the function arguments are pointers. This is typical of functions implemented as Fortran subroutines, which is what the underlying optimization code is designed to work with. Fortran passes all of its arguments by reference, which is accomplished in C and C++ by passing pointers. The first argument is n, the size of the argument of the function. The second argument points to the values for the objective function. This pointer should refer to an array of at least size n which store the n argument values. The result is returned in the memory location pointed to by the third argument.

Upon construction, class `Optimizer` takes arguments for the size of the system and an objective function. To date this class does not implement solutions when the user provides a second or third function for the gradient or Hessian of the objective function.

The complete interface of the `Optimizer` class is given in Appendix D.2. The procedure for using the class is roughly as follows. First, the class is created with a pointer to the objective function it will operate on. The values of various parameters for the optimization are then set, including the minimum and maximum parameter values and diagonal scaling. When all the options are set, the optimization is begun by calling the member function `optimize(void)`. Afterword, the results of the optimization can be retrieved with the member functions `get_result()` and `get_min()`.

6.6 Application Classes: Models, Discretizations and Operators

These classes are part of the third and least abstract level of code developed in this project. The code described in this and the following sections are designed specifically for solving the electromagnetic hysteresis equations described in Chapter 4 in the case with homogeneous material coefficients with or without the supra-conductive backing.

Three distinct versions of this problem are implemented with the classes described

in this section. These are differential equation versions arising from the Debye and Lorentz polarization models and the integro-differential equation version using the hysteresis model of polarization. This last problem has three versions of its own, corresponding to different hysteresis functions. Two implement the functions which arise from the Debye and Lorentz polarization models and the third represents a generic Galerkin approximation of the hysteresis function.

The implementation uses several class families to represent different aspects of the problem. These families are state holder classes, model classes, parameter classes, discretization classes, and operator classes. All three versions of the problem (differential Debye and Lorentz and the integro-differential version) each have a corresponding state holder class, discretization class and operator class. The model class family contains three classes which represent the Debye, Lorentz and spline hysteresis functions are for use with the classes which implement the integro-differential version of the problem. The organization of the code largely reflects the split into representing the original mathematical problem and representing the numerical algorithm as applied to this problem. Complete descriptions of the interfaces of these classes are given in Appendix F

6.6.1 Model Classes

Models are a family of classes which represent hysteresis models of electrical polarization. They accept vector arguments upon construction which contain the values of the parameters appearing in the hysteresis model. An existing model class can also be given new parameter values by calling its `assign` member function. The model class implements the derivative of the kernel function $\dot{g}(\cdot)$ and the function's zero value, $g(0)$ for use by other classes.

Three model classes have been developed. The Debye and Lorentz models are represented by `DebyeModel` and `LorentzModel`. Internally, class `ElementModel` uses the `GalerkinC0` class to represent the hysteresis function through its Galerkin approximation.

6.6.2 Parameter Classes

The Debye and Lorentz model classes of the three model classes have an associated class designed to handle the conversion of parameter values. These are classes `DebyeParams` and `LorentzParams`. These classes convert the physical parameter values into the values used in the numerical implementations of the models. The Debye model is expressed in terms of the physical parameters $(\sigma, \epsilon_\infty, \epsilon_s, \tau)$ while calculations using this model are expressed in terms of the related set of parameters $(\sigma, \epsilon_d, \epsilon_\infty, \lambda)$ where $\epsilon_d = \epsilon_s - \epsilon_\infty$ and $\lambda = 1/c\tau$. (The parameter c is the speed of light in vacuum.) In the Lorentz model, the physical parameters are $(\sigma, \epsilon_s, \epsilon_\infty, \omega_0, \tau)$ while the computations are expressed in terms of $(\sigma, \epsilon_\infty, \hat{\omega}_0, \hat{\omega}_p, \lambda)$ where $\hat{\omega}_0 = \omega_0/c, \hat{\omega}_p = \hat{\omega}_0\sqrt{\epsilon_s - \epsilon_\infty}$ and $\lambda = 1/2c\tau$.

Each of these classes is publicly derived from class `Vector`, which means that they can be passed to functions which expect a vector argument. The values of the computational parameters are stored as the elements of the vector in the correct locations expected by the model classes described above. This means that these classes can be passed to the corresponding model class as the vector of parameters, after converting the physical parameters to the required computational parameters.

6.6.3 Discretization Classes

There are three members of this class family. Classes `DebyeDisc` and `LorentzDisc` represent the differential problems arising from the Debye and Lorentz models and `GeneralDisc` is used to represent the general hysteresis model of polarization. Of these, `DebyeDisc` and `LorentzDisc` are actual classes, while `GeneralDisc` is a class template, where the template argument is a *model* class. Each of these classes computes and stores the matrices and vectors which arise in the Galerkin finite element approximation of their corresponding problems. These appear in the systems of equations (4.10), for the differential Debye problem, (4.22) for the differential Lorentz problem and (4.44) for the hysteresis problem.

All of these classes represent the same problem geometry with a finite slab whose front and back edges correspond to grid nodes in the spatial discretization and all three classes take arguments upon construction which describe the spatial discretization of the problem. These are an object of type `GalerkinC0<uniformGrid>` and two integers for the left and right elements which are on the boundary of the material. (These classes can be easily made into templates on the type of grid, if other grids are desired.) Class `GeneralDisc` takes a second `GalerkinC0<uniformGrid>` object on construction which represents the discretization of the history variable s. (This too could be templated on the type of the grid if desired.)

The two classes representing differential problems, `DebyeDisc` and `LorentzDisc`, also take a `Vector` argument which contains the values of these material parameters while class `GeneralDisc`, accepts an object of a corresponding model class. These classes can also be updated with a new argument of the correct type. All three classes have a member function `assign` which takes a vector argument of new parameter values and updates the problem they represent. Class `GeneralDisc` can also be updated this way with a new object of the appropriate model type. This feature is designed for use in the inverse problem, where the same discretization will be used to represent a succession of simulations which differ in the model parameter values. Notice that the geometric information represented by the discretization class cannot be changed, so the dimensions of the problem and location of the slab are fixed once the object is created.

6.6.4 State Holder Classes

There are three classes in this family, `DebyeStateHolder`, `LorentzStateHolder` and `GeneralStateHolder`, corresponding to the three discretization classes. Their jobs are to represent the different pieces of data which make up the state of these systems in a convenient way. All three members of this family are publicly derived from class `Vector`, which provides them with storage and allows them to be treated as vectors inside of integration routines.

Inside each of these classes, the data is stored in the memory of the `Vector` parent class. For `DebyeStateHolder` and `LorentzStateHolder` the data consists of the state variables e, $\dot{e}$ and p. Class `LorentzStateHolder` also contains the data for $\dot{p}$ which appears in the second order polarization equation. The class `GeneralStateHolder` stores all of the data necessary for the hysteresis formulation of the problem. This consists of the vectors e, $\dot{e}$ and the matrix U for approximating the history of $\dot{e}$.

State holder classes each use *view* vectors internally to provide access to different parts of the data (see Section 6.1.2). For example the variables $e, \dot{e}, p$ and, in `LorentzStateHolder`, $\dot{p}$ are all referenced with view vectors so they can be easily extracted from the rest of the data. Also, the differential equation for the polarization uses only the elements of the variables e and $\dot{e}$ which correspond to positions inside the material domain Ω. These sub-vectors are also accessed through view vectors inside these classes. Similarly, inside `GeneralStateHolder`, e, $\dot{e}$ and the matrix U are accessed through view vectors.

6.6.5 Operator Classes

There are also three members of this class family, `DebyeOperator`, `LorentzOperator` and `GeneralOperator` which correspond to the three members of the discretization class family and the three state holder classes. Like the general discretization class, the General operator class is a template, but its argument is the discretization type and not the model type, i.e., `GeneralDisc<DebyeModel>` would be used with class `GeneralOperator<GeneralDisc<DebyeModel> >`.

An operator class object is permanently associated with a discretization class object when it is created. The operator class object then uses the matrix and vector objects stored in the discretization object to implement the differential equation for the problem it represents. These systems of equations are defined in (4.10), for the differential Debye problem, (4.22) for the differential Lorentz problem and (4.44) for the hysteresis problem.

Since operator classes represent differential equations, they implement the interface described in Section 6.4 for differential equation classes. Internally, the state of the system and derivative are represented through objects of the corresponding state holder class. The operator class also contains a function object of type `Windowed<Harmonic>` representing the interrogating signal. The evaluation of this function and its application to the derivative of the system is done in the course of evaluating the derivative.

The division of each of the different problems into operator and discretization classes is done to separate the storage and implementation of each problem. All of the data arising from the physical parameters and discretization of the problem is stored in the discretization class, while the computation of the state derivative is provided by the operator class. A consequence of this division is that changing the physical parameters of the problem only requires updating the discretization class. This is most useful in the context of the inverse problem, where it is necessary to repeat the simulation with many different parameter values. We note that the object representing the interrogation signal is an exception, since it is contained by the operator class and not the discretization class. This does not effect the usefulness of the separation, however, because the parameters of the interrogating signal are considered to be fixed in the inverse problems and there is no need to be able to update these parameters.

6.7 Performing Simulations

The classes described above are all of the tools necessary to write code which performs simulations of the differential Debye and Lorentz problems and the hysteresis formulation of the problem. An example of code which does so is provided in Appendix G. We outline the general procedure here. All of these steps can be performed in a few lines of code, and many require just one.

1. If desired, load parameter values from data files, or query the user for values.

2. Create the grid and `GalerkinC0` object which represents the spatial discretization.

3. For a hysteresis problem, create the grid and `GalerkinC0` object which represents the discretization of the history.

4. For hysteresis problems, create a model class object from the parameter values.

5. Create an object of the discretization class from the `GalerkinC0` object(s), and the parameter values or model object.

6. Create an object representing the interrogating signal.

7. Create an object of the operator class using the discretization object and the interrogating signal.

8. Create an integrator system class with the operator class object.

9. Advance the system class to the desired time values. Output the results of the integration.

6.8 Forward and Inverse Problems

These are the most application specific classes in the library, since they were written to implement very specific computational problems. The forward problems of particular interest to us are the ones discussed in Sections 4.1, 4.3, 4.5 in which we collect values of the electric field at uniform times at the location $z = 0$. The programs originally written to solve the forward and inverse problem for the Debye model were written in Fortran and are not a part of this body of code. Code for solving forward problems with Debye is included here because it required no special effort once the code for solving the more general problems was in place.

The classes which implement the forward and inverse problems come in pairs which are designed to work together. There are currently two such pairs, both consisting of template classes. Classes `ForwardOperator` and `InverseOperator` are general operator classes which can be used with any system class (see Section 6.4.1). Class `ForwardOperator` is templated on the type of the system class, while the class `InverseOperator` is templated on the type of the forward operator and the type of the discretization.

Classes `ForwardGeneral` and `InverseGeneral` are specificly for implementing the forward and inverse problems for the hysteresis problem of Section 4.5. These are both template classes templated on the type of the model class (`DebyeModel`, `LorentzModel`, `ElementModel`). The advantage of using these less general classes is that the programmer must do less to use them, as will be seen below.

The responsibilities of the forward problem class in each pair are the same. It integrates the system of equations over a series of time steps, extracts the data from the solution at each time step (the value of the electric field at $z = 0$) and returns the data as a Vector. Each version of the forward problem is also updateable, i.e., it can be assigned to a new problem with different material parameters. Forward problem classes have two uses in the code. They can be used on their own to generate the data, which could then be saved to a file or used by an inverse problem object. Also, both of these classes are used by the corresponding inverse problem classes to perform the forward problem when the objective function is evaluated.

The `ForwardOperator` class is constructed by providing a system class object to be integrated, the duration of the simulation, and the number of data points to be collected. The user of the code is responsible for constructing the system object from the appropriate discretization and operator classes. The forward problem can then be updated by updating the discretization class used by the integrable system.

When it is constructed, an object of class `ForwardGeneral` must only be provided an object of class `GeneralDisc`, a `Windowed<Harmonic>` object representing the interrogating signal, plus the interval of time to integrate over and number of data

points. Objects of the `GeneralOperator` class and the integrable system class are automatically created and used inside the `ForwardGeneral` object. The interfaces for these classes are given in Appendix F.5

Both of the inverse classes have two primary responsibilities. First, they both implement the objective function $J(\vec{q})$. Internally, this means adjusting between the argument of the objective function $\vec{q}$ and the full set of model parameters, evaluating the forward problem with the parameter values, then computing and returning the l_2 norm of the error. The first step is necessary because not all of the parameters are optimized over in any given inverse problem. The inverse problem class keeps track of which parameters are being optimized over and fixed values which are being used for the other parameters.

These classes are also responsible for organizing and executing the inverse problem. They keep track of the data used in computing the l_2 error, upper and lower bounds and diagonal scaling for the parameters and the results of the optimization once completed. When the inverse problem is executed, appropriate arguments are passed to an object of class `Optimizer` which performs the optimization. The inverse problem class then interprets and presents the results.

The complete interfaces for these classes are given in Appendix F.6.

Chapter 7

Conclusion

7.1 Summary of Results

We began this work with the derivation of models for the scattering and transmission of electromagnetic radiation by dispersive materials in one dimension. Beginning with Maxwell's equations for macroscopic materials, we imposed constitutive assumptions about the material and derived a coupled system of equations for the electric field and electric polarization in the time domain. We then investigated the analytic and computational properties of these equations using two different formulations of the polarization constitutive laws: a low order differential equation, or a convolution of a kernel function with the history of the electric field. This latter formulation includes the former through a suitable choice of the kernel function in the convolution, as well as higher order differential models and the combined effects of multiple differential models.

The computational investigations detailed in Chapter 4 begin with the first and second order differential polarization models, known as the Debye and Lorentz models. Numerical simulations of the scattering/transmission problem were implemented for both of these equations. The computational results exhibited the behavior known to occur in these systems, specifically the development of the Sommerfeld and Brillouin precursor signals. This duplicates the important results of the frequency domain

analyses of these problems. We then turned to using these simulations in the recovery of the values of the parameters which appear in the polarization and wave equations. Our results in this section show that, even in the presence of noise corrupting the observed data, recovery of parameter values adequate for describing the electrodynamics of the material is possible. Furthermore, the estimation of the parameters of the Debye model was successfully used in a hybrid algorithm for estimating the physical parameters and thickness of the one-dimensional material slab. The ability to recover these parameters in the presence of noise of a magnitude equal to 5% of the data is an important first in problems of this kind. This has significant implications for the use of these electromagnetic scattering techniques in applied problems of material interrogation.

We also investigated the numerical implementation of the hysteretic polarization model. To do this we used a Galerkin approximation of the electric field history and kernel function over a finite history duration. In the case of the Debye model, we saw that it is possible to obtain a highly accurate reconstruction of the results obtained by the differential equation formulation. We also noted that the expense of this computation compared to the differential equation version is considerable. In the case of the second order Lorentz model, we argued that the oscillatory nature of the hysteresis kernel makes a similar computation prohibitively expensive. We also tested the inverse problem for the hysteresis formulation for a kernel function which reproduces the Debye model results, and a general kernel function. In the case of the Debye hysteresis kernel, the estimation problem was shown to give similar results to the estimation problem using the differential model. This is due to the near equality of solutions arising from these methods. In this test, the general kernels were represented through a Galerkin approximation of the hysteresis function's derivative and value at zero (where the hysteresis function derivative is the function in the integral polarization term). This is enough to reconstruct the hysteresis function up to the accuracy of the discretization. We found that a very difficult optimization problem

arises from expressing the kernel as a function of the coefficients of this approximation. To test this, we attempted to reconstruct the Debye model kernel through its coefficients. The full problem of recovering all of the coefficients in the approximation was shown to be over-parameterized, resulting in an optimization which could not be performed in an acceptable amount of computer time. Reducing the size of the problem by optimizing over just the two most significant parameters improved the conditioning of the problem, but was still shown to be inferior to the results obtained by using the Debye model parameterization. This result, combined with the added expense of the hysteresis simulations, and the inability of the hysteresis formulation to handle the Lorentz problem, leads us to conclude that the differential formulation of the problem is more suitable for the estimation of parameters.

Although the hysteresis model did not prove to be more useful than the differential model for simulation or estimation, it proved to be very valuable in the well-posedness results for these problems. In Chapter 3, we showed that the one dimensional scattering and transmission problem is well-posed using the hysteresis formulation of the polarization. That is, we showed the existence, uniqueness, and continuous dependence of the solutions on the data. This was done through the construction of Galerkin style finite-element approximations which were then shown to converge to a solution of the infinite-dimensional problem. A final piece of the analysis, the strong convergence of the finite dimensional Galerkin approximations, is provided in [BZ].

The code written to solve many of the numerical problems arising in this research was also done with the goal of developing a re-usable library of mathematical software. This is described in Chapters 5 and 6. In this attempt we also sought a better understanding of the issues surrounding the development of object-oriented numerical software. The success of this part of the project can best be seen in the development of the application programs used to obtain the results of Chapter 4. After a large initial outlay of time, swift progress was made in creating specific application code. The initial investment of time involved learning more about the C++ programming

language and object-oriented design, plus the development and testing of the fundamental parts of the library. The development of the higher level software modules happened more quickly in conjunction with the first application programs. The final application programs, written after the library had reached near-final form, were developed and coded the most quickly. For example, the code for representing the system of equations arising from the differential Lorentz model was done quickly by adapting the classes already created for the differential Debye model. The new parts of these classes were written using the same mathematical operations as the classes for Debye. The classes performing elementary mathematical operations required no modification at all. Once this was done, no additional code needed to be written to perform forward simulations for Lorentz, since the code implementing it was written to act generically for various simulations. While the software modules for performing inverse problems was originally model specific, the final versions can perform the desired inverse problem for any of the models covered here. An important consequence of this design is that it is readily expandable. Classes written to represent other models not covered here, and even other problems, can be used directly in place of already existing components, allowing much of the existing code to be reused. The most fundamental mathematical operators have the greatest potential for re-use, while the more problem specific application classes are written to allow modification for other problems. This represents the real power of object-oriented software design.

7.2 Further Directions

We suggest further research directions for the computational and software-oriented parts of this project. Our comparison between the differential and hysteretic formulations is currently limited to the two differential models of Debye and Lorentz. Higher order differential models may cause a re-evaluation of the relative efficiency of the two formulations for simulations. This investigation would require a thorough

understanding of the sensitivity of the problem to the polarization model. For example, is it important to whether or not higher order differential equations, whose solutions are expensive to compute, can be effectively approximated by lower order models. This will also require a greater involvement with the intended applications of the simulation, since this will affect the definition of effective approximation.

While any software library can be arbitrarily extended by the addition of new features, there are specific advances which would greatly benefit the development project begun here. First, a better understanding of the performance issues of object-oriented software is called for. While care was taken to retain high efficiency in core computations by reusing Fortran code in this library, the performance shortfall is still considerable. An understanding of the causes and solutions to these performance problems would greatly improve the usefulness of the library. Furthermore, there are other development projects in scientific C++ which produce freely available software. The incorporation of these for the core mathematical operations would improve performance while still allowing for the local development of application specific modules.

List of References

[APM] R. Albanese, J. Penn and R. Medina, Short-rise-time microwave pulse propagation through dispersive biological media, J. of Optical Society of America A, **6** (1989), pp. 1441–1446.

[Bal] C.A. Balanis, *Advanced Engineering Electromagnetics,* J. Wiley & Sons, New York, 1989.

[BSGII] H.T. Banks, D.S. Gillian and V.I. Shubov, Global solvability for damped abstract nonlinear hyperbolic systems, Differential and Integral Equations **10** (1997), pp. 309–332

[BB] H.T. Banks and M.W. Buksas, Electromagnetic interrogation of dielectric materials, Technical Report CRSC-TR98-30, Center for Research in Scientific Computation, North Carolina State University, August, 1998.

[BIW] H.T. Banks, K. Ito and Y. Wang, Wellposedness for damped second order systems with unbounded input operators, Differential and Integral Equations **8** (1995), pp. 587–606.

[BJ] H.T. Banks and M.Q. Jacobs, The optimization of trajectories of linear functional differential equations, SIAM J. Control **8** (1970), pp. 461–488.

[BKo] H.T. Banks and F. Kojima, Boundary shape identification problems in two dimensional domains related to thermal testing of materials, Q. Appl. Math **47** (1989), pp. 273–293.

[BKoW] H.T. Banks, F. Kojima and W.P. Winfree, Boundary Estimation problems arising in thermal tomography, Inverse Problems **6** (1990) pp. 897–921.

[BSW] H.T. Banks, R.C. Smith and Y. Wang *Smart Material Structures: Modeling Estimation and Control,* Masson/J. Wiley, Paris/Chichester, 1996.

[BZ] H.T. Banks and J. Zou, Regularity and approximation of systems arising in electromagnetic interrogation of dielectric materials, Technical Report CRSC-TR98-33, Center for Research in Scientific Computation, North Carolina State University, September 1998.

[BK] R.S. Beezley and R.J. Krueger, An electromagnetic inverse problem for dispersive media, J. Math. Phys. **26** (1985), pp. 317–325.

[BF] J.G. Blaschak and J. Franzen, Precursor propagation in dispersive media from short-rise-time pulses at oblique incidence, J. Opt. Soc. Am. A **12** (1995) pp. 1501–1512

[Bri] L. Brillion, *Wave Propagation and Group Velocity*, Academic, New York, 1960.

[Bui] D.D. Bui, On the well-posedness of the inverse electromagnetic scattering problem for a dispersive medium, Inverse Problems **11** (1995), pp. 835–863.

[Buz] G. Buzzi-Ferraris, *Scientific C++*, Addison Wesley, Reading, Mass. 1993.

[CM] D. Colton and P. Monk, The detection of leukemia by electromagnetic waves, preprint

[CS] J. Corones and Z. Sun, Simultaneous reconstruction of material and transient source parameters using the invariant imbedding method, J. Math. Phys. **34** (1993), pp.1824–1845

[Dat] K. Dattatri. *C++ Effective Object-Oriented Software Construction,* Prentice Hall. Upper Saddle River, NJ. 1996

[Deb] P. Debye, *Polar Molecules*, Chem. Catalog Co., New York, 1929.

[Glo] R. Glowinski *Numerical Methods for Nonlinear Variational Problems,* Springer-Verlag. New York, New York. 1984

[HFL] S.He, P. Fuks and G.W.Larson, An optimization approach to time-domain electromagnetic inverse problem for a stratified dissipative slab, IEEE T. Att. Prop. **44** (1996), pp. 1277–1282.

[HS] S. He and S. Ström, The electromagnetic scattering problem in the time domain for a dissipative slab and a point source using invariant imbedding, J. Math. Phys. **32** (1991), pp. 3529–3539.

[Jac] J. D. Jackson, *Classical Electrodynamics,* 2nd Ed., J. Wiley & Sons, New York, 1975.

[Kri] K.L. Kreider, Time dependent direct and inverse electromagnetic scattering for the dispersive cylinder, Wave Motion **11** (1989), pp. 427–440.

[KK] G. Kristenson and R.J. Krueger, Direct and inverse scattering in the time domain for a dissipative wave equation. I. Scattering operators, J. Math. Phys. **27** (1986), pp. 1667–1682

[KK2] G. Kristenson and R.J. Krueger, Direct and inverse scattering in the time domain for a dissipative wave equation. II. Simultaneous reconstruction of dissapation and phase velocity, J. Math. Phys **27** (1986), pp.1683–1693

[KK3] G. Kristenson and R.J. Krueger, Direct and inverse scattering in the time domain for a dissipative wave equation. III. Scattering operators in the presence of a phase velocity mismatch, J. Math. Phys **28** (1987), pp.360–370.

[KK4] G. Kristenson and R.J. Krueger, Direct and inverse scattering in the time domain for a dissipative wave equation. Part 4. Use of phase-velocity mismatches to simplify inversions, Inverse Problems **5** (1989), pp.375–388.

[Kru] R.J. Krueger, An inverse problem for a dissipative hyperbolic equation with discontinuous coefficients, Q. Appl. Math **34** (1976), pp. 129–147.

[Kru2] R.J. Krueger, An inverse problem for an absorbing medium with multiple discontunities, Q. Appl. Math **36** (1978), pp.235–253.

[Kru3] R.J. Krueger, Numerical Aspects of a dissipative inverse problem, IEEE Trans. Att. Prop. AP-29 (1981), pp.253–261.

[Lad] S. Ladd, *C++ Templates and Tools*, 2nd Ed. M&T Books, New York. 1996.

[Ler] I. Lerche, Some singular, nonlinear integral equations arising in physical problems, Q. Appl. Math. **44** (1986), pp.319–326.

[Lions] J.L. Lions, *Optimal Control of Systems Goverend by Partial Differential Equations*, Springer-Verlag, New York, 1971.

[Mur] R. Murray, *C++ Strategies and Tactics*, Addison Wesley, Reading Mass. 1993

[Oua] S. Oualine, *Practical C++ Programming*, O'Reilly & Associates, Sebastopol, CA. 1995

[Pi] O. Pironneau, *Optimal Shape Design for Elliptic Systems*, Sprinver-Verlag, New York, 1983.

[PTVF] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipies in Fortran*, 2nd Ed. Cambridge University Press, Cambridge, 1994.

[Ree] S. Reek, *Pointers on C*, Addison Wesley, Reading, Mass. 1998.

[RGKM] V.G. Romanov, J. Gottlieb, I. Kabankhin and S.V. Martakov, An inverse problem for special dispersive media arising from ground penetrating radar, J. Inv. Ill-Posed Problems **5** (1997), pp.175–192.

[St] J.A. Stratton, *Electromagnetic Theory,* McGraw-Hill, New York, 1941.

[Str] B. Stroustrup, *The C++ Progrmming Language*, 3rd Ed. Addison Wesley, Reading, Mass. 1997.

[Sun] Z. Sun, Direct scattering and reconstruction of internal sources, Wave Motion **16** (1992), pp.249–263.

[VH] A.R. von Hippel, *Dielectric Materials and Applications*, J. Wiley, New York, 1954.

[Wloka] J. Wloka, *Partial Differential Equations*, Cambridge University Press, Cambridge, 1987.

[Wes] V.H. Weston, On the inverse problem for a hyperbolic dispersive partial differential equation, J. Math. Phys. **13** (1972), pp. 1952–1956.

[Wes2] V.H. Weston, Factorization of the dissipative wave equation and inverse scattering, J. Math. Phys. **29** (1988), pp.2205–2218.

[Wol] L.V. Wolfersdorf, On an electromagnetic inverse problem for dispersive media, Q. Appl. Math. **49** (1991), pp.237–246.

Resources available on the World Wide Web:

[Diff] Diffpack Public Access Release 1.4, `http://www.nobjects.com/Public/`, Numerical Objects AS.

[LAP++] LAPACK++, `http://math.nist.gov/lapack++/`, National Institute of Science and Technonogy.

[MTL] Matrix Template Library `http://www.lsc.nd.edu/research/mtl/` Laboratory for Scientific Computing.

[TNT] Template Numerical Toolkit, `http://math.nist.gov/tnt/` National Institute of Science and Technonogy.

APPENDICES

Appendix A

Interface Specification for the Matrix/Vector Library

A.1 Class Matrix

Implementation:

Include: `Matrix.hh`

Link: library `libmatrix.a` (`-lmatrix`)

Constructors:

`Matrix(void)` Create a size zero matrix.

`Matrix(int m, int n)` Create a matrix of size m by n.

`Matrix(const Matrix&B)` Create a copy of the matrix B.

`Matrix(double *p, int m, int n)`

Construct a *view* matrix of the data at pointer p.

`Matrix(const double* p, int m, int n)`

Construct a matrix containing a copy of the data at pointer p.

Operators:

`void resize(int m, int n)`

Resize the matrix to dimensions m by n. Discard old contents.

`void resize(const Matrix& B)`

Resize to match dimensions of matrix B. Do not copy contents of B.

`void copy(const Matrix&)`

Adopt the size of and copy the data in Matrix B

`void view(Matrix& B)`

Become a *view* of the data in matrix B. Retain current dimensions.

`void view(double* p, int m, int n)`

Become a matrix *view* of the data pointed to by `p` with dimensions m by n.

`void operator=(double d)`

Usage: `M=d;` Assign the scalar value d to each element in the matrix.

`void operator=(const Matrix& B)`

Usage: `A=B;` Adopt the size of and copy the data in matrix B.

`double& operator()(int i, int j)`

Usage: `A(i,j);` Returns a reference to the $i^{\text{th}}, j^{\text{th}}$ element of the matrix. Can be used for access or assignment, i.e., `v = A(0,0);` or `A(i,j)=1/(1+i+j);`

Information:

`int get_size(void)`

Return total size (area) of matrix.

`int get_size(int i)`

Return size of dimension i=0,1.

`int get_lda(void)`

Return leading dimension of storage (=`get_size(0)`)

`double* addr(void)`

Return pointer to data.

`bool owns_data(void)`

Return true if matrix owns data, false if it is a view.

A.2 Class Vector

Implementation:

Include: `Vector.hh`

Link: library `libmatrix.a` (`-lmatrix`)

Constructors:

`Vector(void)`

Construct a size zero vector.

`Vector(int m, bool s=`*`true`*`)`

Construct a row or column vector of size m. When the second argument is `true` or no second argument is given, a column vector is created.

`Vector(const Vector& B)`

Construct a copy of the vector B.

`Vector(double* v, int m, bool s=`*`true`*`)`

Construct a *view* vector of the data at the pointer `v`.

`Vector(const double* v, int m, bool s=`*`true`*`)`

Construct a vector containing a copy of the data at pointer `v`.

Operators: All of these operators are directly related to the same operator defined for class `Matrix` in Appendix A.1. Operators which define a size for a `Vector` object

also have an optional final boolean argument which indicates a row vector when *false*. The other difference is that access through a single index is supported.

```
void resize(int n, bool s=true)
void resize(const Vector& B)

void copy(const Vector& B)
void view(double* v, int m, bool s=true)
void view(Vector& B)

void operator=(const Vector& B)

void operator=(double s)

double& operator()(int i)
double& operator()(int i, int j)
```

Information:

`int get_size(void)`

Return the total size of the vector. This is equal to the size of its non-unit dimension.

`int get_size(int i)`

Return the size of dimension $i = 0, 1$. One of these two sizes is 1.

A.3 Class BandedMatrix

Implementation:

Include: `BandedMatrix.hh`

Link: library `libmatrix.a` (`-lmatrix`)

Constructors:

`BandedMatrix(void)`

Construct a banded matrix of size zero.

`BandedMatrix(int n, int du, int dl)`

Construct a banded matrix with size n, du upper bands and dl lower bands.

`BandedMatrix(const BandedMatrix& B)`

Construct a copy of banded matrix B.

Operators:

`void copy(const BandedMatrix& B)`

`void operator=(const BandedMatrix& B)`

Copy contents of B. Resize if necessary.

`void operator=(double d)`

Set all elements in the bands of the matrix to the scalar value d.

`void resize(int n, int du, int dl)`

Resize the banded matrix to size n with du upper and dl lower diagonals. Discard old contents.

`void resize(const BandedMatrix& B)`

Resize the banded matrix to match the dimensions of banded matrix B. Do not copy the contents of B.

`double& operator()(int i, int j)`

Usage: `B(i,j);` Returns a reference to the $i^{\text{th}}, j^{\text{th}}$ element, if this is inside the bands of the matrix. Can be used for reference or assignment. If outside the bands, return zero and ignore assignments.

Information:

`int get_size(int i)`

Return logical size of dimension i.

`int get_size(void)`

Return area of storage matrix.

`int get_dl(void)`

Return number of lower diagonals.

`int get_du(void)`

Return number of upper diagonals.

`int get_lda(void)`

Return leading dimension of storage matrix. This is equal to $du + 2 * dl + 1$.

`double* addr(void)`

Return address of storage matrix (including extra storage).

`double* stor_addr()`

Return address of element where matrix storage begins. This skips over the memory locations in the extra storage bands required for factorizations.

A.4 Class MatrixLU

The matrix factorization class `MatrixLU` is designed to implement LU factorizations of square matrices represented by the class `Matrix`. It cannot be generally resized or have its elements assigned to, hence many of the constructors and operators implemented for matrices are not present here.

Implementation:

Include: `MatrixLU.hh`

Link: library `libmatrix.a` (`-lmatrix`)

Constructors:

`MatrixLU(void)`

Construct a `MatrixLU` object that does not contain the factorization of a matrix.

`MatrixLU(const MatrixLU& B)`

Construct a `MatrixLU` object that copies the data from and factors the contents of the matrix B.

Operators:

`void copy(const Matrix& B)`

Copy the data from matrix B and factor. Resize as necessary.

`void copy(const MatrixLU& B)`

Become a copy of the `MatrixLU` object B.

`int solveIP(Matrix &B)`

Compute solution to linear equation X=MB, where M is represented by the MatrixLU object. Return the result in B.

A.5 Class BandedMatrixLU

Like `MatrixLU`, `BandedMatriLU` contains the data for an LU factorization of a banded matrix represented by a `BandedMatrix` object. It also is not generally resizeable and its individual elements can not be accessed directly.

Implementation:

Include: `BandedMatrixLU.hh`

Link: library `libmatrix.a` (`-lmatrix`)

Constructors:

`BandedMatrixLU(void)`

Construct a `BandedMatrixLU` object that does not contain the factorization of a banded matrix.

`BandedMatrixLU(const BandedMatrixLU& B)`

Construct a `BandedMatrixLU` object which copies the data from the argument factors the contents of the banded matrix B.

Operators:

`void copy(const BandedMatrix& B)`

Resize copy data from and factor contents of B.

`void copy(const BandedMatrixLU& B)`

`int solveIP(Matrix &B)`

Compute solution to linear equation X=MB, where M is represented by the BandedMatrixLU object. Return the result in B.

Appendix B

Blas Routines

Implementation:

Include: `blas1++.hh`, `blas2++.hh`, `blas3++.hh` and `blas4++.hh` as necessary

Link: library `libmyblas.a` (`-lmyblas`)

B.1 BLAS 1

`double Blas_Norm1(Matrix x)` $|x|_1$

`void Blas_Add_Mult(Matrix y, double a, const Matrix x)` $y \leftarrow ax + y$

`void Blas_Mult(Matrix y, double a, Matrix x)` $y \leftarrow ax$

`void Blas_Copy(Matrix y, Matrix x)` $y \leftarrow x$

`double Blas_Dot_Prod(Matrix x, Matrix y)` $x^T y$

`double Blas_Norm2(Matrix x)` $|x|_2$

`void Blas_Scale(double a, Matrix x)` $x \leftarrow ax$

`void Blas_Swap(Matrix x, Matrix y)` $x \leftrightarrow y$

`int Blas_Index_Max(Matrix x)` i such that $|x_i| = \max_j |x_j|$

`void Blas_Apply_Plane_Rot(Vector x, Vector y, double c, double s)`

`void Blas_Gen_Plane_Rot(double a, double b, double c, double s)`

B.2 BLAS 2

`void Blas_Mat_Trans_Vec_Mult(A, x, y, alpha, beta)` $y \leftarrow \alpha A^T x + \beta y$

`A Matrix`

`x,y Vector`

`alpha double` default = 1.0

`beta double` default = 1.0

`void Blas_Mat_Vec_Mult(A, x, y, alpha, beta)` $y \leftarrow \alpha A x + \beta y$

`A Matrix`

`x,y Vector`

`alpha double` default = 1.0

`beta double` default = 1.0

`void Blas_Mat_Vec_Mult(A, x, y, alpha, beta)` $y \leftarrow \alpha A x + \beta y$

`A BandedMatrix`

`x,y Vector`

`alpha double` default = 1.0

`beta double` default = 1.0

`void Blas_R1_Update(A, x, y, alpha)` $A \leftarrow \alpha A x y^T + A$

`A Matrix`

`x,y Vector`

`alpha double` default = 1.0

B.3 BLAS 3

`void Blas_Mat_Mat_Mult(A, B, C, alpha, beta)` $C \leftarrow \alpha A B + \beta C$

`A,B,C Matrix`

`alpha double` default = 1.0

`beta double` default = 0.0

`void Blas_Mat_Trans_Mat_Mult(A, B, C, alpha, beta)` $C \leftarrow \alpha A^T B + \beta C$

`A,B,C Matrix`

`alpha double` default = 1.0

`beta double` default = 0.0

`void Blas_Mat_Mat_Trans_Mult(A, B, C, alpha, beta)` $C \leftarrow \alpha A^T B + \beta C$

`A,B,C Matrix`

`alpha double` default = 1.0

`beta double` default = 0.0

B.4 Extensions to the BLAS

These functions are declared in file `blas4++.hh`

`void multiply_column(A, B, C, i, alpha, beta)` $C^{(i)} \leftarrow \alpha A B^{(i)} + \beta C^{(i)}$

`A BandedMatrix`

`B,C Matrix`

`i int`

`alpha double` default = 1.0

`beta double` default = 0.0

$C^{(i)}$ refers to the ith column of matrix C. Performs `Blas_Mat_Vec_Mult` for banded matrices with $x = B^{(i)}$ and $y = C^{(i)}$.

`void Blas_Mat_Mat_Mult(A, B, C, alpha, beta)` $C \leftarrow \alpha A B + \beta C$

`A BandedMatrix`

`B,C Matrix`

`alpha double` default = 1.0

`beta double` default = 0.0

Appendix C

Function Classes

C.1 Elementary Function Classes

All of the following classes implement `operator()(double)` for evaluation. One or more parameters is given at construction. Only `Polynomial` supports the changing or parameter values after construction.

`Constant`

Represents the constant function c, where c is specified at construction.

`Polynomial`

Represents the arbitrary polynomial $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$. There are two constructors for this class.

`Polynomial(int)` Sets the degree to the argument. Defaults to zero.

`Polynomial(int, double*)` Sets the degree and takes coefficient values from the double pointer.

The degree can be changed with the function `set_degree(int)` and retrieved by `get_degree(void)`. The coefficients a_i are accessed through *`name`*`[i]` where `i` is an integer index in the range 0 through the degree.

`Harmonic`

Represents $A\sin(2\pi \times f + \phi)$. The constructor is `Harmonic(double, double, double)` where the three arguments are A, f and ϕ.

`Indicator`

Represents the indicator function $I_{x_1,x_2}(x)$ which is 1 for $x_1 \leq x \leq x_2$ and zero otherwise. The constructor is `Indicator(double, double)` where the arguments are x_1 and x_2.

`Windowed`

This is a template applied to other classes to window their output to a specific interval. Its specification is `Windowed<F>` where `F` is the type of the class it is applied to. Its two constructors are `Windowed(f, double, double)` and `Windowed(f,ind)`. In each, `f` is the object of function class `F`. The pair of double values are the lower and upper limits of the window, or the argument `ind` is an `indicator` object representing the window.

Appendix D

Other Library Routines

D.1 Quadrature of Differential Equations

D.1.1 Class rkMethodIP

The `rkMethodIP` class represents the application of the classical 4th Order Runga-Kutta method (see [PTVF, p705]) to a particular differential equation. It also keeps track of the current state of the system, including the state vector and the current time. These quantities can be manipulated though the appropriate operator member functions describe below.

Implementation:

Include: `rkMethodIP.hh`

Declare as: `rkMethodIP<F>` where `F` is a differential equation class type.

Constructors:

`rkMethodIP(F &func)`

Construct a `rkMethodIP` object to integrate function `F`.

`rkMethodIP(F &func, Vector &v)`

Construct a `rkMethodIP` object to integrate function `F` beginning from the state in vector `v`.

Operators:

```
void set_time(double value)
void set_state(Vector value)
```

Manipulate the time and state of the system represented by the `rkMethodIP` class.

```
void set_step_size(double value)
```

Set the step size used in the Runga-Kutta algorithm.

```
void step()
void step(int number)
void integrate_to(double final_time)
void advance(double amount)
```

Perform various integrations of the system: A single step, a specified number of steps, to a particular time, or forward a particular amount.

Information:

```
int get_size(void)
```

Returns the size of the system of differential equations.

```
double get_time(void)
Vector& get_state(void)
```

Returns the current time and state of the system. Notice that `get_state` returns a reference, so this can be used for assignment of values in the state also.

```
double get_step_size(void)
```

Returns the step size used in the Runga-Kutta implementation.

D.1.2 Classes lsodaMethod and SystemIntegrator

`SystemIntegrator` is a generic class which represents the combination of a differential equation class with a method for integrating it. At this time, `lsodaMethod` is the only integrator class designed to work with `SystemIntegrator` so we document these classes together. Since `lsodaMethod` is an abstract base class, no instances of this class are created. Its purpose is to provide functions which implement the method to `SystemIntegrator`

Implementation:

Include: `lsodaMethod.hh` and `SystemIntegrator.hh`

Link: `lsodaMethod.o`

Declare as: `SystemIntegrator<F,lsodaMethod>` where `F` is a differential equation class.

Constructors:

`SystemIntegrator(int n, F& function)`

Construct a `SystemIntegrator` object with a state of size n for integrating the given function. (Note that n must match the size of the system operated on by `function`.)

Operators:

`void set_time (double value)`

`void set_state(Vector value)`

Set the time and state of the system being integrated.

`void advance(double amount)`

`void integrate_to(double time_stop)`

Advance the state of the system by the specified amount, or to the specified time.

Information:

`int get_size(void)`

Return the size of the system being integrated.

`double get_time(void)`

`Vector& get_state(void)`

Return the time and state of the system being integrated. Note that `get_state` returns a reference, so this function can be used for setting the state also.

D.2 Class Optimizer

Implementation:

Include: `Optimizer.hh`

Link: `Optimizer.o, OPTmine.o`

Constructors:

`Optimizer(int n, ObjectiveType func)`

Create an object representing the optimizer applied to the function `func`. The first argument, n, is the number of arguments of the objective function and `ObjectiveType` is shorthand name (created with the command `typedef`) for the actual type `void (*)(int*, double*, double*)` This type is a pointer to a function with no return argument and which accepts three arguments, a pointer to integer, and two pointers to double. This is the type of the objective function and is suitable for use with Fortran subroutines. The objective function's first argument is the number of parameters, the second contains the values of the parameters and the third is a return argument which contains the result.

Operators:

`void set_min(Vector)`

```
void set_max(Vector)
void set_diag(Vector)
void set_init(Vector)
void set_method(int)
void optimize(void)
```

These functions set the minimum and maximum allowed values, the diagonal scaling and the initial values for each parameter. The final function is called to begin the optimization once these parameters have been set.

Information:

```
int get_size(void)
int get_error(void)
double get_min(void)
Vector get_result(void)
Vector get_gradient(void)
```

Function `get_size` returns the size of the system being optimized (n). The others are for use after the optimization is completed. They return the status code reported by the core routine, the resulting objective value, the arguments where this value is obtained and the estimated gradient at that point.

Appendix E

Finite-Element Classes

E.1 Grid Classes

Grid classes represent the division of a one dimensional interval into subintervals. There are currently two grid classes, `grid` and `uniformGrid`. The class `grid` is more general, while `uniformGrid` only represents grids where the subintervals all have the same size. This allows class `uniformGrid` to implement some functions more efficiently. Aside from the constructors, the interfaces of the classes are identical.

Implementation:

Include: `grid.hh` or `uniformGrid.hh`

Link: `grid.o` or `uniformGrid.o`

Constructors:

`grid(double* v, int n)`

Forms a grid using the values from the memory locations pointed to by `v`. There are `n` nodes in all, with the lowest and highest grid points being `v[0]` and `v[n-1]`.

`uniformGrid(int n, double low, double high)`

Forms a uniform grid of points between the values `low` and `high`. There are $n-2$ interior nodes and the distance between them is $\frac{h-l}{n-1}$ where h, l are given in arguments `high` and `low`.

Information:

```
int size(void)
```

```
int intervals(void)
```

Return the number of nodes n and number of sub-intervals $n-1$.

Operators:

```
double operator()(int i)
```

Return the coordinate of node number i, where $i = 0$ is the left end and $i = n-1$ is the right end of the interval.

```
double d(int i)
```

Return the length of the i^{th} sub-interval or the distance between nodes $i+1$ and i.

```
int locate(double x)
```

Return the number of the interval containing point x. Returns -1 if x is not in the interval of the grid.

E.2 Class ShapeFunctionsC0

Implementation:

Include: `ShapeFunctionsC0.hh`

Link: `ShapeFunctionsC0.o`

Constructors:

`ShapeFunctionsC0::ShapeFunctionsC0(int)`

Construct a `ShapeFunctionsC0` object representing the shape functions of degree $d = 1, 2, 3$.

Operators:

`double gridpoint(int n)`

Returns the location of gridpoint n in the reference interval $[-1, 1]$.

`double shape(double x, int n)`

`double operator()(double arg, int n)`

Evaluate the n^{th} shape function at x: $_n(x)$.

`double deriv(double x, int n)`

Evaluates the derivative of the n^{th} shape function at x: $s'_n(x)$.

`double M(int i, int j)`

`double K(int i, int j)`

`double W(int i, int j)`

Evaluate the product of basis functions over interval `n`. `M` is the interval of $s_i s_j$, `W` is $s'_i s_j$ and `K` is $s'_i s'_j$.

`const double* Mmatrix(void)`

`const double* Kmatrix(void)`

`const double* Wmatrix(void)`

Access the matrices `M`, `K` and `W` through pointers to their first elements.

E.3 Class BasisFunctionsC0

Implementation:

Include: `BasisFunctionsC0.hh`

Constructors:

`BasisFunctionsC0(int degree, int domains, bool left, bool right)`

Construct an object representing the basis functions of degree d on n domains. True values for the boolean arguments cause the basis functions to be zero on the corresponding boundary of the domain.

Information:

`int system_size(void)`

`int domain_num(void)`

Operators:

`int FirstFunction(int n)`

`int LastFunction(int n)`

Return the first and last functions whose supports include the interval `n`.

`int FirstDomain(int n)`

`int LastDomain(int n)`

Return the first and last domains which make up the support of function `n`.

`double operator()(double x, int i, int j)`

`double deriv(double x, int i, int j)`

Evaluate $s_k(x)$ and $s'_k(x)$, where s_k is the shape function which makes up the part of basis function i on interval j.

`double M(int i, int j, int n)`

`double K(int i, int j, int n)`

`double W(int i, int j, int n)`

Evaluate the product of basis functions over interval `n`. `M` is the interval of $\phi_i\phi_j$, `W` is $\phi'_i\phi_j$ and `K` is $\phi'_i\phi'_j$. The result is not scaled to any particular geometry.

`const double* Mmatrix(void)`

`const double* Kmatrix(void)`

`const double* Wmatrix(void)`

Access the matrices `M`, `K` and `W` through pointers to their first elements.

E.4 Class GalerkinC0

Implementation:

Include: `GalerkinC0.hh`

Declare as: `GalerkinC0<GridType>` where `GridType` is the type of a grid class.

Constructors:

`GalerkinC0<GridType>(G g, int d, bool left, bool right)`

Construct an object representing Galerkin finite elements of degree `d` on the grid represented by the grid object`g`. Boolean arguments `left` and `right` are true to indicate basis functions which are zero on the corresponding boundary.

Information:

`int intervals(void)`

`int system_size(void)`

`int get_degree(void)`

Operators:

`int FirstFunction(int n)`

`int LastFunction(int n)`

Return the first and last basis functions whose supports include interval `n`.

`double M(int i, int j, int n)`

`double W(int i, int j, int n)`

`double K(int i, int j, int n)`

Return the appropriate integral of basis functions on the interval `n`. `M` is the interval of $\phi_i\phi_j$, `W` is $\phi_i'\phi_j$ and `K` is $\phi_i'\phi_j'$

`double operator()(double x, int n)`

`double deriv(double x, int n)`

Evaluate $\phi_n(x)$ and $\phi_n'(x)$.

`double left(int n)`

`double right(int n)`

Left and right boundaries of the support of basis function ϕ_n.

`bool left_zero(void)`

`bool right_zero(void)`

Return `true` if basis elements are constrained to be zero on that boundary.

E.5 Function Classes for use with GalerkinC0

The template argument `E` refers to the `GalerkinC0<GridType>` type. Template argument `I` refers to an index class. It must support `double operator()(int i)`. Template argument `F` is a function class and must support `double operator()(double)`.

`void ConstMass<E>(E&, BandedMatrix&)`

`void ConstFlex<E>(E&, BandedMatrix&)`

`void ConstStiffness<E>(E&, BandedMatrix&)`

Create a mass, flex, or stiffness matrix where the contributions from each interval in the domain are weighted at unity. Return the result in the `BandedMatrix` argument.

`void AssembleMass<E,I>(E&, I&, BandedMatrix&)`

`void AssembleFlex<E,I>(E&, I&, BandedMatrix&)`

`void AssembleStiffness<E,I>(E&, I&, BandedMatrix&)`

Create a mass, flex, or stiffness matrix where the contributions from each interval in the domain are weighted according to the values in the Index class I. I is typically a `Vector`.

`void InnerProduct<E,F>(E&, F&, Vector&)`

Compute each element in $\langle \phi_i, f \rangle$, where the $\phi_i(\cdot)$ are represented by the element class and $f(\cdot)$ is represented by the class `F`. Return the result in the `Vector` argument.

`MatrixProduct<E,F>(E&, F&, Vector&)`

This is a class constructed with arguments for `E` and `F`. Member function `double prod(int i, int j)` returns $\langle f\phi_i, \phi_j \rangle$ where the $\phi_i(\cdot)$ are represented by the element class and f is represented by the `F` argument.

Appendix F

Application Classes

F.1 Parameter Classes

Parameter classes are used in conjunction with the model classes `DebyeModel` and `LorentzModel`. They convert the parameter which appear in the polarization equations into the parameters which appear in the numerical implementations. Both of these classes are derived from class `Vector` and so can be used in place of vector arguments. This is most useful with the model classes which take vector arguments for specifying the parameters.

F.1.1 DebyeParams

Implementation:

Include: `DebyeParams.hh`

Constructors:

`DebyeParams(double sigma, double eps_s, double eps_inf, double tau)`

Construct a `DebyeParams` object from the parameter values σ, ϵ_s, ϵ_∞ and τ. An optional boolean fifth argument can be passed which, if true, indicates that the fourth argument is actually $\lambda = 1/c\tau$ and not τ.

Operators

`double& sigma(void)`

`double& eps_d(void)`

`double& eps_inf(void)`

`double& psi(void)`

Access, through a reference, each of the parameter values used in the numerical implementations. Note that in the code, `psi` is used for λ.

F.1.2 LorentzParams

Implementation:

Include: `DebyeParams.hh`

Constructors:

`LorentzParams(double sigma, double eps_s, double eps_inf, double tau)`

Construct a `DebyeParams` object from the parameter values σ, ϵ_s, ϵ_∞, ω_0 and τ. An optional boolean sixth argument can be passed which, if true, indicates that the fifth argument is actually $\lambda = 1/2c\tau$ and not τ.

Operators

`double& sigma(void)`

`double& eps_inf(void)`

`double& omega_0(void)`

`double& omega_p(void)`

`double& psi(void)`

Access, through a reference, each of the parameter values used in the numerical implementations. Note that in the code, `psi` is used for λ.

F.2 Model Classes

F.2.1 DebyeModel

Implementation:

Include: `DebyeModel.hh`

Constructors:

`DebyeModel(const Vector& params)`

Constructs a Debye model class using parameter values from the argument. The order of the parameters in the argument should be $(\sigma, \epsilon_d, \epsilon_\infty, \lambda)$, where $\epsilon_d = \epsilon_s - \epsilon_\infty$ and $\lambda = 1/c\tau$.

Information:

`int get_size(void)`

Operators:

`void assign(const Vector& params)`

Causes the model object to adopt the parameters in the argument.

`double operator()(double s)`

Evaluates the hysteresis kernel function at s.

`double g(void)`

`double sigma(void)`

`double eps_inf(void)`

`double eps_d(void)`

`double psi(void)`

Return the values of the various parameters in the Debye model. Note that in the code, `psi` is used for the parameter λ.

F.2.2 LorentzModel

Implementation:

Include: `LorentzModel.hh`

Constructors:

`LorentzModel(const Vector& params)`

Constructs a Lorentz model class using parameter values from the argument vector. The order of the parameters in the vector should be $(\sigma, \epsilon_\infty, \omega_0, \omega_p, \lambda)$.

Information:

`int get_size(void)`

Operators:

`void assign(const Vector& params)`

Causes the model object to adopt the parameters in the argument.

`double operator()(double s)`

Evaluates the hysteresis kernel function at s.

`double g(void)`

`double sigma(void)`

`double eps_inf(void)`

`double psi(void)`

`double omega_p(void)`

`double nu(void)`

Return the values of the various parameters in the Lorentz model. Note that in the code, `psi` is used for λ.

F.2.3 ElementModel

Class `ElementModel` is templated on the type of basis function elements it uses to represent the kernel function. Currently, this is always `GalerkinC0<uniformGrid>` for compatibility with other classes. An object of this class contains the values of the coefficients in the function approximation, plus the values of σ, ϵ_∞ and $g(0)$.

Implementation:

Include: `ElementModel.hh`

Declare as: `ElementModel<BasisType>` where `BasisType` is the type of basis elements.

Constructors:

`ElementModel(BasisType& elements, double eps_inf, double g, double sigma)`

Construct with a basis elements object and values for ϵ_∞, g_0 and σ.

`ElementModel(BasisType& elements)`

Construct with a basis elements object only.

Information:

`int get_size(void)`

Return the total number of parameters in the model. This is equal to the number of coefficients in the basis elements approximation, plus three.

Operators:

`assign(const Vector & val)`

Assign the elements in the vector argument to the coefficients and other parameters in the `ElementModel` object.

`void assign_coeff(const Vector& val)`

Assign just the coefficients of the basis elements.

`void set_eps_inf(double arg)`

`void set_g(double arg)`

`void set_sigma(double arg)`

Assign values to the other parameters.

`double operator()(double s)`

Evaluates the hysteresis kernel function at s.

`Vector& get_coeffs(void)`

`double eps_inf(void)`

`double g(void)`

`double sigma(void)`

Returns the values of the various parameters. Notice that `get_coeffs` returns a reference. This means that this function can also be used to assign to the coefficients.

F.3 Discretization Classes

F.3.1 DebyeDisc

Implementation:

Include: `DebyeDisc.hh`

Constructors:

`DebyeDisc(Vector& params, BasisType& basis, int left, int right)`

Construct the discretization object given a vector of parameters, a basis elements object of type `GalerkinC0<uniformGrid>`, and the value of the left and rightmost basis functions which are non-zero inside the material domain.

Operators:

`assign(Vector& params)`

Adopt the parameter values in the argument.

Information:

`double get_psi(void)`

`double get_eps(void)`

`double get_sigma(void)`

Return the values of the physical parameters.

`int L(void)`

`int R(void)`

Return the numbers of the left and rightmost basis elements in the material sub-domain.

`int get_size(void)`

Return the size of the elements e and $\dot{e}$ in the system

`int get_sub_size(void)`

Return the number of elements in the sub-domain. Equal to result of `R()-L()`+1.

`int get_total_size(void)`

Return the total size of the differential system. Equal to 2×`get_size()`+ `get_sub_size()`.

`int intervals(void)`

Return the number of intervals in the basis elements.

`int get_num_params(void)`

`Vector& get_params(void)`

Return and access the number of parameters. For objects of this class, the number of parameters is always four.

Class `DebyeDisc` also contains publicly accessible data members which store the coefficients used by the `DebyeOperator` class.

F.3.2 LorentzDisc

Implementation:

Include: `LorentzDisc.hh`

Constructors:

```
LorentzDisc(Vector& lorentz, GalerkinC0<uniformGrid>& elements,
int left, int right)
```

Construct the discretization object given a vector of parameters, a basis elements object of type `GalerkinC0<uniformGrid>`, and the value of the left and rightmost basis functions which are non-zero inside the material domain.

Operators:

```
assign(Vector& params)
```

Adopt the parameter values in the argument.

Information:

```
double get_psi(void)
double get_sigma(void)
double get_omega_0(void)
double get_omega_p(void)
```

Return the values of the physical parameters.

```
int L(void)
```

`int R(void)`

Return the numbers of the left and rightmost basis elements in the material sub-domain.

`int get_size(void)`

Return the size of the elements e and $\dot{e}$ in the system

`int get_sub_size(void)`

Return the number of elements in the sub-domain. Equal to `R()-L()`+1.

`int get_total_size(void)`

Return the total size of the differential system. Equal to 2×`get_size()`+ `get_sub_size()`.

`int intervals(void)`

Return the number of intervals in the basis elements.

`int get_num_params(void)`

`Vector& get_params(void)`

Return and access the number of parameters. For objects of this class, the number of parameters is always five.

Class `LorentzDisc` also contains publicly accessible data members which store the coefficients used by the `LorentzOperator` class.

F.3.3 GeneralDisc

Implementation:

Include: `GeneralDisc.hh`

Declare as: `GeneralDisc<Model>` where `Model` is one of `DebyeModel`, `LorentzModel` or `ElementModel`.

Constructors:

`GeneralDisc(Model& model, GalerkinC0<uniformGrid>& space_elements, int left, int right, GalerkinC0<uniformGrid>& time_elements)`

Construct the discretization object given a model object of the templated type, a basis elements object of type `GalerkinC0<uniformGrid>`, and the value of the left and right-most basis functions which are non-zero inside the material domain, plus another `GalerkinC0<uniformGrid>` object describing the discretization of the space variable.

Operators:

`assign(Vector& params)`

Adopt the parameter values in the argument.

`assign(Model& model)`

Adopt the parameters of the physical model given by the model class.

Information:

`int L(void)`

`int R(void)`

Return the numbers of the left and rightmost basis elements in the material sub-domain.

`int get_size(void)`

Return the size of the elements e and $\dot{e}$ in the system

`int get_sub_size(void)`

Return the number of elements in the sub-domain. Equal to result of `R()-L()`+1.

`int get_time_size(void)`

Return the number of elements in the history approximation.

`int get_total_size(void)`

Return the total size of the differential system. Equal to 2×`get_size()`+ `get_sub_size()`×`get_time_size()`.

`int intervals(void)`

Return the number of intervals in the basis elements.

`int get_num_params(void)`

`Vector& get_params(void)`

Return and access the number of parameters. For objects of this class, the number of parameters is always five.

F.4 Operator Classes

F.4.1 DebyeOperator

Implementation:

Include: `DebyeOperator.hh`

Constructors:

`DebyeOperator(const Windowed<Harmonic> &signal, DebyeDisc& disc)`

Construct a `DebyeOperator` object with an object representing the interrogating signal and a `DebyeDisc` object.

Operators:

`void operator()(double time, Vector &state, Vector &deriv)`

Evaluate the derivative of the system at time `time` and state `state`. Return the result in the vector `deriv`.

F.4.2 LorentzOperator

Implementation:

Include: `LorentzOperator.hh`

Constructors:

`LorentzOperator(const Windowed<Harmonic> &signal, LorentzDisc& disc)`

Construct a `LorentzOperator` object with an object representing the interrogating signal and a `LorentzDisc` object.

Operators:

`void operator()(double time, Vector &state, Vector &deriv)`

Evaluate the derivative of the system at time `time` and state `state`. Return the result in the vector `deriv`.

F.4.3 GeneralOperator

Implementation:

Include: `GeneralOperator.hh`

Constructors:

`GeneralOperator(const Windowed<Harmonic> &signal, LorentzDisc& disc)`

Construct a `GeneralOperator` object with an object representing the interrogating signal and a `LorentzDisc` object. The type `sigType` is a short name for class `Windowed<Harmonic>`.

Operators:

`void operator()(double time, Vector &state, Vector &deriv)`

Evaluate the derivative of the system at time `time` and state `state`. Return the result in the vector `deriv`.

F.5 Forward Classes

F.5.1 ForwardOperator

Implementation:

Include: `ForwardOperator.hh`

Declare as: `ForwardOperator<System>` where `System` is a class type which supports the interface for an integrable system.

Constructors:

`ForwardOperator(System& sys, double time, int number)`

Construct a `ForwardOperator` object to integrate the provided system `sys`. The data is collected at `number` points at uniform intervals between zero and `time`.

Information:

`int get_size(void)`

Get the size of the system being integrated.

`Vector& get_state(void)`

Access the current state of the system being integrated.

Operators:

`void evaluate(Vector& result)`

Integrate the system of equations from zero to the time specified at construction. Return the value of the first coefficient of e at each step in the vector argument.

F.5.2 ForwardGeneral

Implementation:

Include: `ForwardGeneral.hh`

Declare as: `ForwardGeneral<ModelType>` where `ModelType` is `DebyeModel`, `LorentzModel` or `ElementModel`.

Constructors:

`ForwardGeneral(DiscType& disc, const Windowed<Harmonic>& signal, double time, int number)`

Construct a `ForwardGeneral` object from a discretization and interrogation signal. The data is collected at `number` points at uniform intervals between zero and `time`. Type `DiscType` is `GeneralDisc<ModelType>` and `sigType` is `Windowed<Harmonic>`.

Information:

`int get_size(void)`

Return the size of the system being integrated.

`int get_num_params(void)`

Return the number of parameters of the model.

Operators:

`void set_time_step(double arg)`

Set the time step for integration of the system.

`void adopt(const ModelType& aModel)`

`void adopt(const Vector& params)`

Adopt a new model, expressed either as a model class or vector of parameters.

`void evaluate(Vector& arg)`

Integrate the system of equations from zero to the time specified at construction. Return the value of the first coefficient of e at each step in the vector argument.

F.6 Inverse Classes

F.6.1 InverseOperator

Implementation:

Include: `InverseOperator.hh`

Declare as: `InverseOperator<ForwardOp,Disc>` where `ForwardOp` the type exact type of a forward operator class described above and `Disc` is the type of the discretization class.

Constructors:

`InverseOperator(ForwardOp&, Disc&, const int*)`

Construct an `InverseOperator` object with the given forward operator object and discretization object. Note that the discretization object must be associated with the forward problem object. The third argument is a pointer to an array of integers containing ones and zeros, which indicates which parameters to optimize. A one in the ith position of this array indicates that the ith parameter is to be optimized.

Information:

`Vector get_result(void)`

Return the parameters resulting from the optimization.

`double get_min(void)`

Returns the minimum residual obtained by the optimization.

`double get_result_code(void)`

Return the status code returned by the optimizer.

Operators:

`void set_data(Vector arg)`

Set the data which the inverse problem attempts to match by optimizing parameter values.

```
void evaluate_data(Vector& params)
```

Evaluate the forward problem at with the given parameter values. Store the result as the data to be reconstructed in the inverse problem.

```
set_initial(const Vector& arg)
void set_min (const Vector& arg)
void set_max (const Vector& arg)
void set_diag (const Vector& arg)
void set_sizes (const Vector& arg)
void set_fixed (const Vector& arg)
void set_scale (const Vector& arg)
void set_method(int i)
```

Set the values of other parameters which influence the optimization. Note that all of the vector arguments refer to the full set of parameters. If the optimization is performed over a subset of the parameters, the correct values from these arguments are selected and passed to the optimizer.

```
void set_active(const int* arg)
```

Reset the list of active parameters. The argument is a pointer to integers containing ones and zeros. A one in the ith place of this array indicates that the ith parameter is to be optimized.

```
double evaluate_opt(Vector& params)
```

Evaluate the objective function at the indicated parameters. Note that this is a reduced set of parameters which contains only values for the parameters being optimized over.

```
void optimize(void)
```

Begin the optimization.

F.6.2 InverseGeneral

Implementation:

Include: `InverseGeneral.hh`

Declare as: `InverseGeneral<Model>` where `Model` is one of the three model types.

Constructors:

`InverseGeneral(const ForwardType&, const int*)`

Construct an `InverseGeneral` object from a forward problem. The type `ForwardType` is `ForwardGeneral<ModelType>`. The third argument is a pointer to an array of integers containing ones and zeros, indicating which parameters to optimize. A one in the ith position of this array indicates that the ith parameter is to be optimized.

Information:

`Vector get_result(void)`

Return the parameters resulting from the optimization.

`double get_min(void)`

Returns the minimum residual obtained by the optimization.

`double get_result_code(void)`

Return the status code returned by the optimizer.

Operators:

`void set_data(Vector arg)`

Set the data which the inverse problem attempts to match by optimizing parameter values.

`void evaluate_data(Vector& params)`

Evaluate the forward problem using the given parameter values. Store the result as the data to be reconstructed in the inverse problem.

`set_initial(const Vector& arg)`

`void set_min (const Vector& arg)`

`void set_max (const Vector& arg)`

`void set_diag (const Vector& arg)`

`void set_fixed (const Vector& arg)`

`void set_method (int i)`

Set the values of other parameters which influence the optimization. Note that all of the vector arguments refer to the full set of parameters. If the optimization is performed over a subset of the parameters, the correct values from these arguments are selected and passed to the optimizer.

`void set_active(const int* arg)`

Reset the list of active parameters. The argument is a pointer to integers containing ones and zeros. A one in the ith place of this array indicates that the ith parameter is to be optimized.

`double evaluate_opt(Vector& params)`

Evaluate the objective function at the indicated parameters. Note that this is a reduced set of parameters which contains only values for the parameters being optimized over.

`void optimize(void)`

Begin the optimization.

Appendix G

Example Programs

G.1 Step through a simulation of the differential Debye model

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include "DebyeParams.hh"
#include "DebyeStateHolder.hh"
#include "DebyeOperator.hh"
#include "uniformGrid.hh"
#include "GalerkinC0.hh"
#include "Vector.hh"
#include "BandedMatrix.hh"
#include "Harmonic.hh"
#include "Windowed.hh"
#include "rkMethodIP.hh"
#include "blas1++.hh"
#include "const.hh"
#include "Options.hh"
#include "Numerics.hh"
#include "Physics.hh"

/*************************************************************

  stepDiffDebye
```

```
  =============

  Step through a simulation of the differential debye model.


**************************************************************/

void main(int argc, char** argv){

  Options options(argc, argv);
  Numerics numerics(options.numerics_name);
  Physics  physics (options.physics_name);

  int space_degree   = numerics.space_degree;
  int space_nodes    = numerics.N_elements;
  bool left_zero     = false;
  bool right_zero    = true;

  //-------------------------------------------------
  // Create Galerkin object
  //-------------------------------------------------

  uniformGrid space(space_nodes, 0, physics.length);
  typedef GalerkinC0<uniformGrid> spaceDisc;
  spaceDisc galerk(space, space_degree, left_zero, right_zero);

  int left  = numerics.left_element;
  int right = numerics.right_element - 1;

  cout << "Left Domain: " << left << endl
       << "Right Domain: " << right << endl;
  cout << "Left edge of material: "  << space(left) << endl;
  cout << "Right edge of material: " << space(right+1) << endl;

  int sys_size = galerk.system_size();
  int domains  = galerk.intervals();

  cout << "Number of basis functions: " << sys_size << endl;
  cout << "Number of domains: "         << domains  << endl;
```

```
// --------------------------------
// Construct debye parameters from
// the loaded parameter values
// --------------------------------

DebyeParams params;
params.assign(physics.sigma, physics.params[0],
      physics.epsilon_r, physics.params[1]);

cout << "Debye parameters: " << endl;
cout << params.sigma()   << endl;
cout << params.eps_inf() << endl;
cout << params.eps_d()   << endl;
cout << params.psi()     << endl;

// --------------------------------
// Construct a Debye Discretization
// --------------------------------

DebyeDisc  debye_disc(params, galerk, left, right);

DebyeStateHolder state(debye_disc.get_size(),
                       debye_disc.L(), debye_disc.R());
state = 0.0;

typedef Windowed<Harmonic> sigType;
double s_final = physics.signal_stop;
double freq    = physics.omega;
Harmonic carrier_signal(freq*2*PI*ETA, freq, PI/2);
sigType input_signal(carrier_signal, 0.0, s_final);

cout << "Creating a Debye Operator: " << endl;
DebyeOperator debye_op(input_signal, debye_disc);

cout << "Constructing rkMethodIP. " << endl;
rkMethodIP<DebyeOperator> problem(debye_op, state);
problem.set_step_size(numerics.time_step);

cout << "Using time step: " << numerics.time_step << endl;
```

```
  // ------------------------------
  // Iterate through the simulation
  // ------------------------------

  cout << "Saving state in file: " << options.data_name << endl;
  double advance = 0.5;
  while (advance>0){

    cout << "Current time: " << problem.get_time() << endl;
    cout << "Advance by: ";
    cin  >> advance;
    problem.advance(advance);

    ofstream out_plot(options.data_name.c_str());
    state.view(problem.get_state());
    out_plot << state.e << endl;
    out_plot.close();

  }
}
```

G.2 Step Through a Simulation of the Hystersis Debye Model

```
#include <iostream.h>
#include <fstream.h>
#include "Matrix++.hh"
#include "Galerkin.hh"
#include "Windowed.hh"
#include "Harmonic.hh"
#include "DebyeParams.hh"
#include "DebyeModel.hh"
#include "GeneralDisc.hh"
#include "uniformGrid.hh"
#include "ForwardGeneral.hh"
#include "const.hh"
```

```
#include "Options.hh"
#include "Numerics.hh"
#include "Physics.hh"

/*************************************************************

  stepHystDebye
  =============

  Step through a simulation of the debye hysteresis model.


*************************************************************/

void main(int argc, char **argv){

  Options options(argc, argv);
  Numerics numerics(options.numerics_name);
  Physics  physics (options.physics_name);

  //-----------------------------------------------
  // Create Galerkin objects
  //-----------------------------------------------

  double r = numerics.history_duration;   // Duration of history.

  int  space_degree   = numerics.space_degree,
       time_degree    = numerics.time_degree,
       space_nodes    = numerics.N_elements,
       time_nodes;
  int  time_intervals = numerics.M_elements;
  bool left_zero      = false;
  bool right_zero     = true;

  time_nodes = time_intervals + 1;

  cout << "History grid size: " << r/time_intervals << endl;

  uniformGrid space(space_nodes, 0, physics.length);
  typedef GalerkinC0<uniformGrid> spaceDisc;
```

```
spaceDisc galerk_space(space, space_degree, left_zero, right_zero);

uniformGrid history(time_nodes, -r, 0);
typedef GalerkinC0<uniformGrid> timeDisc;
timeDisc galerk_hist(history, time_degree, false, false);

int left  = numerics.left_element;
int right = numerics.right_element - 1;

cout << "Left Domain: " << left << endl
     << "Right Domain: " << right << endl;
cout << "Left edge of material: "  << space(left) << endl;
cout << "Right edge of material: " << space(right+1) << endl;

// ---------------------------------
// Construct debye parameters from
// the loaded parameter values
// ---------------------------------

DebyeParams params;
params.assign(physics.sigma, physics.params[0],
              physics.epsilon_r, physics.params[1]);

// --------------------
// Create a DebyeModel
// --------------------

cout << "Constructing a Debye Model..." << endl;
DebyeModel model(params);

// ---------------------------------------
// Create a Discretization with the Model
// ---------------------------------------

typedef GeneralDisc<DebyeModel> DiscType;
DiscType debye_disc(model, galerk_space, left, right, galerk_hist);

// Create a stateholder from the discretization.

GeneralStateHolder state(debye_disc.get_space_size(),
```

```
                          debye_disc.L(), debye_disc.R(),
 debye_disc.get_time_size());
state = 0.0;

// ------------------
// Construct a Signal
// ------------------

typedef Windowed<Harmonic> sigType;
double s_final = physics.signal_stop;
double freq    = physics.omega;
Harmonic carrier_signal(freq*2*PI*ETA, freq, PI/2);
sigType input_signal(carrier_signal, 0.0, s_final);

// ----------------------------
// Construct a General Operator
// ----------------------------

cout << "Creating a General Operator... " << endl;
typedef GeneralOperator<DiscType> OpType;
OpType debye_op(input_signal, debye_disc);

cout << "Constructing rkMethodIP... " << endl;
rkMethodIP<OpType> problem(debye_op, state);
problem.set_step_size(numerics.time_step);

cout << "Using time step: " << numerics.time_step << endl;

// -----------------------------
// Iterate through the simulation
// -----------------------------

// Temporairly modified for timing.

cout << "Saving state in file: " << options.data_name << endl;

double advance = 0.5;
while (advance>0){

  cout << "Current time: " << problem.get_time() << endl;
```

```
    cout << "Advance by: ";
    cin  >> advance;
    problem.advance(advance);

    ofstream out_plot(options.data_name.c_str());
    state.view(problem.get_state());
    out_plot << state.e << endl;
    out_plot.close();

  }
}
```